**Министерство образования Республики Беларусь**

**Учреждение образования**
**«Гомельский государственный технический университет имени П. О. Сухого»**

**Кафедра «Информационные технологии»**

**И. А. Мурашко, Д. А. Литвинов**

# БАЗЫ ЗНАНИЙ И ПОДДЕРЖКА ПРИНЯТИЯ РЕШЕНИЙ В СИСТЕМАХ АВТОМАТИЗИРОВАННОГО ПРОЕКТИРОВАНИЯ

**КУРС ЛЕКЦИЙ**
**по одноименной дисциплине для студентов специальности 1-40 01 02 «Информационные системы и технологии (по направлениям)» дневной формы обучения**

**Электронный аналог печатного издания**

**Гомель 2011**

УДК 004.8(075.8)
ББК 328.13я73
М91

*Рекомендовано к изданию научно-методическим советом факультета автоматизированных и информационных систем ГГТУ им. П. О. Сухого*
(*протокол № 8 от 28.03.2011 г.*)

Рецензент: зав. каф. «Промышленная электроника» ГГТУ им. П. О. Сухого канд. техн. наук, доц. *Ю. В. Крышнев*

**Мурашко, И. А.**

М91 Базы знаний и поддержка принятия решений в системах автоматизированного проектирования : курс лекций по одноим. дисциплине для студентов специальности 1-40 01 02 «Информационные системы и технологии (по направлениям)» днев. формы обучения / И. А. Мурашко, Д. А. Литвинов. – Гомель : ГГТУ им. П. О. Сухого, 2011. – 83 с. – Систем. требования: PC не ниже Intel Celeron 300 МГц ; 32 Mb RAM ; свободное место на HDD 16 Mb ; Windows 98 и выше ; Adobe Acrobat Reader. – Режим доступа: http://lib.gstu.local. – Загл. с титул. экрана.

ISBN 978-985-535-005-8.

Предназначен для формирования представлений о проблемах искусственного интеллекта, методах представления и хранения знаний, применения возможностей искусственного интеллекта в решении прикладных задач конструкторского и технологического проектирования. Изложение материала проиллюстрировано примерами на языках программирования систем искусственного интеллекта Лисп и Пролог.

Для студентов специальности 1-40 01 02 «Информационные системы и технологии (по направлениям)» дневной формы обучения.

**УДК 004.8(075.8)**
**ББК 328.13я73**

**ISBN 978-985-535-005-8**

## Содержание

# ВВЕДЕНИЕ

В настоящее время широко применяются программные средства, базирующиеся на технологиях и методах искусственного интеллекта (экспертные системы, системы поддержки принятия решений, системы интеллектуального анализа данных и т. п.) [4], [7], [8]. По мнению специалистов [10], в ближайшее время экспертные системы будут играть ведущую роль на всех этапах жизненного цикла любого продукта. Уже сейчас они активно используются при проектировании, изготовлении и сопровождении изделий.

Создание экспертных систем значительно расширило область приложения вычислительной техники, позволив использовать ее для решения неформализованных задач [4]. Различие между формализованными и неформализованными задачами обусловливается характером знаний, используемых для решения этих задач. Знания, которыми владеет специалист в какой-либо области, можно разделить на точные, выраженные формально, и неточные (неформальные). Точные знания формулируются в книгах и руководствах в виде общих, универсальных и строгих суждений, законов, формул, моделей, алгоритмов и т. д. Неформальные знания обычно не попадают в книги и руководства вследствие их субъективности и «неточности». Знания этого рода являются результатом обобщения многолетнего опыта работы специалиста (эксперта) и представляют собой, как правило, множество эвристических приемов и правил. Они образуют то, что называют опытом и интуицией специалистов. Экспертные системы предназначены, в первую очередь, для решения неформализованных задач на основе неточных знаний, представляющих опыт эксперта по решению задач в предметной области [11].

Можно выделить два типа экспертных систем. Первый тип – это системы, предназначенные для подъема общего уровня профессионализма в некоторой предметной области (проектирование, обучение, медицине и др.). Такие системы выступают, как правило, в роли справочной системы, в которой хранятся ответы на все случаи жизни. Другими словами, в системе накоплена информация, извлеченная из различных источников знаний, и сохранен опыт ведущих специалистов в данной области (экспертов). Это массовые и недорогие системы, доступные всем пользователям. Обычно базы знаний систем такого типа являются замкнутыми, решатель имеет встроенные функции, которыми управляет некоторый сценарий общения с поль-

зователем, а работа самого пользователя не требует от него никаких специальных знаний.

Второй тип экспертных систем рассчитан на специалистов высокой квалификации. Эти системы исполняют роль ученика при работе специалиста. Они снабжают его оперативной информацией, делают технические расчеты, предлагают специалисту альтернативные варианты решения и т. д. Такие системы играют роль мощного специфического инструмента, помогающего специалисту сделать свою деятельность более эффективной.

Широкое использование компьютеров во всех сферах деятельности привело к накоплению огромных объемов информации в каждой предметной области. По оценкам специалистов, объемы хранимой информации удваиваются каждые два-три года [3]. Причем только очень малая ее часть будет когда-либо просмотрена человеческим глазом. Поэтому выход из данной ситуации заключается в широком применении технологий инженерии знаний для поиска и извлечения новых полезных знаний из накопленной информации (в базах данных и знаний).

# 1. ИНТЕЛЛЕКТУАЛЬНЫЕ СИСТЕМЫ

## 1.1. Искусственный интеллект

### *1.1.1. Основные понятия искусственного интеллекта*

Термин «искусственный интеллект» (ИИ) появился в 80-е гг. XX в. Не существует единого и общепринятого определения ИИ, так как нет и универсального определения человеческого интеллекта [10]. К ИИ принято относить ряд алгоритмов и программных систем, которые могут решать некоторые задачи так, как это делает человек.

***Искусственный интеллект*** (с лат. *intellectus* – познание) – раздел информатики, изучающий методы, способы моделирования и воспроизведения с помощью ЭВМ различной деятельности человека, связанной с решением каких-либо задач.

***Интеллектуальной*** называется система, позволяющая усилить интеллектуальную деятельность человека, за счет ведения с ним осмысленного диалога.

*Цель ИИ* – смоделировать разумную деятельность человека, автоматизировать мышление.

*Область применения ИИ* – решение слабо формализованных задач, обладающих следующими свойствами:

1. Цели задач не могут быть представлены в виде математических функций.

2. Алгоритмы достижения цели не могут быть описаны строго математически (не существует алгоритмического решения задачи).

3. Существует алгоритмическое решение, но пространство поиска решения очень велико.

4. Для решения задач требуются эвристики – утверждения, основанные на опыте, интуиции, с целью найти более рациональное решение, а не оптимальное, путем исключения заранее непригодных решений.

5. Знания, используемые для решения задач, обладают следующими свойствами: не полные, ошибочные, разнородные, неоднозначные, противоречивые, динамичные.

***Естественный интеллект*** (ЕИ) – способность осмысленно приобретать, воспроизводить, использовать знания, понимать конкретные и абстрактные идеи, постигать отношения между идеями и объектами.

*Цель ЕИ* – воспроизвести черты ЕИ: мышление, инстинкты (врожденные программы), безусловные и условные рефлексы, моторика, сенсорика.

*Свойства ЕИ*:

1. Способность к классифицированию по шаблону: кластеризация, категоризация, распознавание.

2. Адаптивность, обучаемость.

3. Способность к дедуктивному мышлению – переход от общего к частному.

4. Способность к индуктивному логическому выводу (от частного к общему).

5. Анализ и синтез (*CASE* – технологии).

6. Способность разрабатывать концептуальные модели.

7. Способность понимать – видеть отношения в задачах и оценивать их в решениях.

### *1.1.2. История и направления развития систем искусственного интеллекта*

Исторически сложились три основных направления в моделировании ИИ [13]. В рамках *первого подхода* объектом исследований являются структура и механизмы работы мозга человека, а конечная цель заключается в раскрытии тайн мышления. Необходимыми этапами исследований в этом направлении являются построение моделей на основе психофизиологических данных, проведение экспериментов с ними, определение механизмов интеллектуальной деятельности.

*Второй подход* в качестве объекта исследования рассматривает ИИ. Здесь речь идет о моделировании интеллектуальной деятельности с помощью вычислительных машин. Целью работ в этом направлении является создание алгоритмического и программного обеспечения вычислительных машин, позволяющего решать интеллектуальные задачи не хуже человека.

*Третий подход* ориентирован на создание смешанных человеко-машинных, или интерактивных, интеллектуальных систем, на симбиоз возможностей естественного и искусственного интеллекта. Важнейшими проблемами в этих исследованиях является оптимальное распределение функций между естественным и искусственным интеллектом и организация диалога между человеком и машиной.

На рис. 1.1 представлены основные направления исследований в области ИИ.

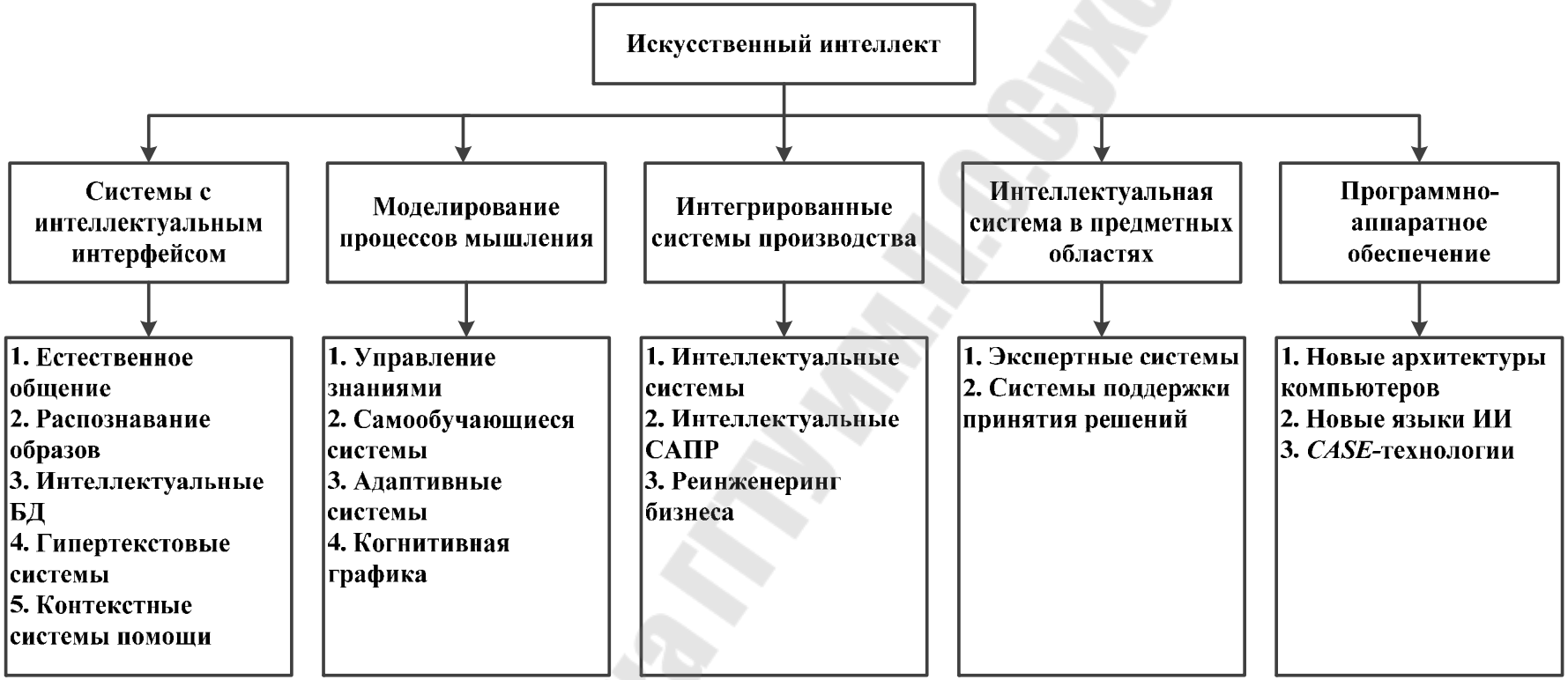



*Рис. 1.1*. Современные направления исследований в области ИИ

В 90-е гг. в исследованиях по ИИ выделились шесть основных направлений:

1. ***Представление знаний***. В рамках данного направления решаются задачи, связанные с формализацией и представлением знаний в памяти интеллектуальной системы (ИС). Для этого разрабатываются специальные модели представления знаний и языки для описания знаний, создаются процедуры и методы, с помощью которых ИС может приобретать знания.

2. ***Манипулирование знаниями***. В рамках рассматриваемого направления строятся способы пополнения знаний на основе их неполных описаний, разрабатываются процедуры обобщения знаний и формирования на их основе абстрактных понятий, создаются методы достоверного и правдоподобного вывода на основе имеющихся знаний, предлагаются модели рассуждений, опирающихся на знания и имитирующих особенности человеческих рассуждений. Манипулирование знаниями тесно связано с представлением знаний.

3. ***Общение***. В круг задач данного направления входят: проблема понимания связных текстов на ограниченном и неограниченном естественном языке, синтез связных текстов, понимание речи и синтез речи, теория моделей коммуникации между человеком и ИС. К этому направлению относится проблема формирования объяснения действий ИС, которые она должна уметь порождать по просьбе человека. На основе исследований в данном направлении формируются методы построения лингвистических процессоров, вопросно-ответных систем, диалоговых систем и других ИС, целью которых является обеспечение комфортных условий для общения человека с ИС.

4. ***Восприятие***. Данное направление традиционно включает: проблемы анализа трехмерных сцен, разработку методов представления информации о зрительных образах в базе знаний (БЗ), создание методов перехода от зрительных сцен к их текстовому описанию и методов обратного перехода, создание средств порождения зрительных сцен на основе внутренних представлений в ИС.

5. ***Обучение***. Предполагается, что ИС будут способны к обучению – решению задач, с которыми они ранее не встречались. Для этого необходимо создать методы формирования условий задачи по описанию проблемной ситуации или по наблюдению за этой ситуацией, научиться переходу от известных решений частных задач к решению общей задачи, создать методы декомпозиции исходной задачи на более мелкие, известные ИС, разработать модели самого процесса обучения, создать теорию подражательного поведения.

6. ***Поведение***. Так как ИС должны действовать в некоторой окружающей среде, то необходимо разработать специальные поведенческие процедуры, для адекватного взаимодействия с окружающей средой, другими ИС и людьми. Для этого следует создать: модели целесообразного поведения, нормативного поведения, ситуативного поведения, специальные методы многоуровневого планирования и коррекции.

Таким образом, все направления развития исследований в области ИИ можно объединить в два основных направления, различающихся подходами к моделированию ЕИ:

1-е направление – ***нейробионическое***. Сущность: моделирование структур и процессов биологического прототипа – головного мозга человека.

2-е направление – ***информационное*** (черный ящик). Сущность: не нужно моделировать структуру мозга, необходимо познать только внешние проявления работы человеческого мозга (правила, закономерности).

### *1.1.3. Классификация систем искусственного интеллекта*

Все существующие ИС делятся на два класса: системы общего назначения и специализированные системы.

***Системы общего назначения*** содержат метапроцедуры поиска, с помощью которых генерируют и исполняют процедуры решения новых конкретных задач. Технология использования таких систем состоит в следующем. Пользователь (эксперт) формирует знания (данные и правила), описывающие выбранное приложение. Затем на основании этих знаний, заданной цели и исходных данных метапроцедуры системы генерируют и исполняют процедуру решения конкретной задачи. Данную технологию называют технологией инженерии знаний. В настоящее время единственными системами общего назначения можно считать оболочки экспертных систем.

К ***специализированным*** относятся системы, которые выполняют решение фиксированного набора задач, предопределенного при проектировании ИС. Сейчас многие специализированные ИС, в частности, речевого общения и обработки изображений стали разрабатывать в виде экспертных систем [16].

На основе исследований в области ИИ сформировалась новая область индустрии – производство интеллектуальных систем.

***Интеллектуальными системами*** называют такие автоматизированные системы, которые предназначены для реализации интеллектуальных задач, решаемых людьми.

К системам ИИ относятся следующие типы автоматизированных систем:

1) естественно-языковые системы;
2) системы речевого общения;
3) системы обработки визуальной информации;
4) системы машинного перевода;
5) экспертные системы и оболочки экспертных систем.

Основными функциями ***естественно-языковых систем*** (ЕЯ-систем) являются: ведение диалога, понимание и генерация высказываний. Классы ЕЯ-систем:

1. *Интеллектуальные вопросно-ответные системы* – разработка моделей и методов, позволяющих осуществлять перевод ЕЯ-высказываний, относящихся к узким и заранее фиксированным областям, в формальное представление, а также обратный перевод.

2. *Системы общения с базами данных* (БД) – обеспечение доступа к информации в БД широкому классу неподготовленных конечных пользователей, которые формулируют запрос к БД на естественном языке.

3. *Диалоговые системы решения задач* – решение заранее определенных классов задач (планирование действий, путешествий, перевозок и т. д.). Разбиение задачи на подзадачи и распределение решения выполняет диалоговая система. Основным направлением практического использования является реализация ЕЯ-общения с экспертными системами;

4. *Системы обработки связных текстов* – предназначены для обработки больших объемов текстовой информацией для извлечения необходимых сведений.

***Системы речевого общения*** предназначены для реализации ЕЯ-общения на основе устной речи. Решают проблемы синтеза, анализа и распознавания речи.

***Системы обработки визуальной информации*** решают три типа задач: обработка изображений (исходные данные и результаты обработки представлены в виде изображений), анализ изображений (входные данные являются изображением, а результат представляется в неизобразительной форме, например, в виде текста), синтез изображений (на входе имеется описание изображения, а на выходе по нему строится само изображение).

***Системы машинного перевода*** предназначены для быстрого доступа к информации на иностранном языке, перевода больших потоков текстов.

***Экспертные системы*** (ЭС) предназначены для решения неформализованных задач, т. е. задач, решаемых с помощью неточных знаний, которые являются результатом обобщения многолетнего опыта работы и интуиции специалистов. Вычислительные возможности в первую очередь определяются БЗ ЭС и только во вторую очередь определяются используемыми методами. Экспертные системы используются при решении задач следующих типов: принятие решений в условиях неопределенности (неполноты), интерпретация символов и сигналов, предсказание, диагностика, конструирование, планирование, управление, контроль и т. д.

## 1.2. Системы искусственного интеллекта

### *1.2.1. Архитектура систем искусственного интеллекта*

Систему принято считать интеллектуальной, если в ней реализованы три группы основных функций:

1) представления и обработки данных;

2) рассуждения;

3) общения с пользователем на естественном языке.

Приведенные обобщенные функции реализуются совокупностью процедур:

*1 группа*: накопление знаний о предметной области; классификация знаний; структурирование знаний; автоматическое поддержание БЗ при ее пополнении; получение и обработка знаний от экспертов.

*2 группа*: инициализация процессов получения новых знаний; соотнесение новых знаний со старыми; пополнение знаний с получением логического вывода, отражающего закономерности в предметной области и накопленных знаниях; обобщение знаний на основе более частных знаний; логическое планирование своей деятельности; осуществление вывода на основе рассуждений по аналогии и т. п.

*3 группа*: общение на естественном языке (или подмножестве профессионального языка); обучение; адаптация; формирование по запросу пользователя объяснений своей деятельности.

Обобщенная и упрощенная структурно-функциональная схема систем ИИ приведена на рис. 1.2. Архитектура систем ИИ определяется функциями конкретного состава задач и их связями между собой. В приведенной схеме можно выделить два основных блока: машина

БЗ и модуль вывода. Машина БЗ реализует первую функцию систем ИИ – представление и обработку знаний – и состоит из блоков:

– ***база фактов*** – содержит факты, носящие конкретный характер, опыт;

– ***база правил*** – содержит элементарные выражения отражающие закономерности, причинно-следственные связи, называемые в теории ИИ продукциями.

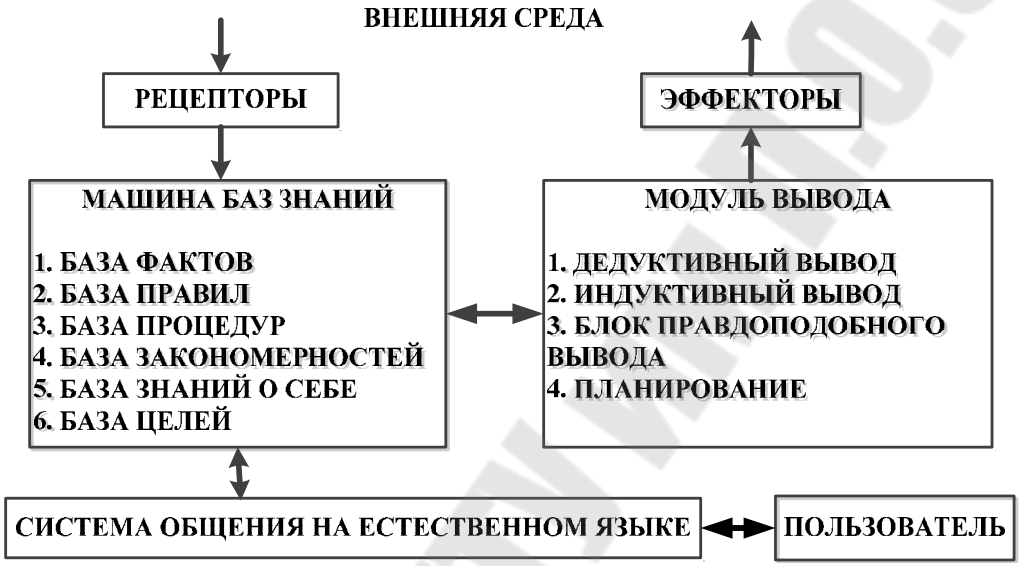



*Рис. 1.2*. Обобщенная структурно-функциональная схема систем ИИ

***База процедур*** – процедуры, выполняющие все необходимые вычисления, преобразования и другие нужные системе последовательности внутренних действий.

***База закономерностей*** – содержит сведения, относящиеся к особенностям предметной области, в которой будет функционировать система (законы, эмпирические зависимости).

***База знаний о себе*** – содержит списки того, что хранится в текущий момент в остальных базах: сведения о единицах информации; сведения о взаимодействии частей системы; сведения о том, как получено решение задачи. То есть описываются возможности и способы функционирования системы.

***База целей*** – содержит целевые структуры, позволяющие организовать процессы движения от исходных фактов, закономерностей, правил к достижению цели. База целей – это способ представления знаний, ориентированный на связи объектов между собой через сообщения, что позволяет реализовывать стратегию вывода.

Это является главным отличием систем ИИ от систем обработки данных (СОД). В традиционной СОД схема передачи управления и использования данных предопределена в программе, т. е. ветвление имеет место в заранее выбранных точках. В системах ИИ ходом рассуждения *управляют данные*, а правила дают возможность оценить ситуацию и предпринять действия. Таким образом, база фактов – это база данных, а база правил, закономерностей и целей составляют основу базы знаний предметной области.

***Модуль вывода*** реализует вторую функцию систем ИИ – функцию рассуждений и состоит из следующих блоков:

1. ***Дедуктивный вывод*** – реализует рассуждения, с помощью которых на основании общих закономерностей, фактов и правил вывода выводятся новые факты.

2. ***Индуктивный вывод*** – применяется для организации вывода новых знаний на основе обобщений отдельных понятий и фактов.

3. ***Блок правдоподобного вывода*** – реализует контроль ошибок в процессе вывода.

4. ***Блок планирования*** – планирует процесс вывода в зависимости от конкретной ситуации.

### *1.2.2. Характеристики знаний*

Основной проблемой, решаемой в системах ИИ всех типов, является проблема *представления знаний*. Информация представляется в компьютере в процедурной и декларативной форме. В процедурной форме представлены программы, в декларативной – данные. В системах ИИ возникла концепция, которая объединила в себе черты как процедурной, так и декларативной информации. Перечислим основные характеристики знаний:

1. ***Внутренняя интерпретируемость***. Каждая информационная единица имеет уникальное имя. Когда данные в памяти лишены имен, отсутствует возможность их идентификации системой, их может извлечь лишь программа по указанию программиста. Что скрывается за тем или иным двоичным кодом, системе было неизвестно. При переходе к знаниям в памяти формируется информация о некоторой *протоструктуре информационных единиц* и *словари данных.* Протоструктура представляет собой специальное машинное слово, в котором указано, в каких разрядах хранится информация. В словарях перечислены имеющиеся в памяти данные. Каждая единица информации будет экземпляром протоструктуры.

2. ***Структурированность***. Информационные единицы должны обладать гибкой, рекурсивной структурой. Каждая информационная единица может быть включена в состав любой другой и из каждой можно выделить некоторые ее составляющие. То есть должна существовать возможность произвольного установления между отдельными информационными единицами отношений типа «часть – целое», «род – вид» или «элемент – класс».

3. ***Связность***. Между информационными единицами возможно установление связей различного типа, характеризующих отношения между ними. Все отношения можно разделить на четыре категории: *отношения структуризации* (задают иерархию информационных единиц), *функциональные отношения* (несут процедурную информацию, позволяющую вычислять одни информационные единицы через другие), *каузальные отношения* (задают причинно-следственные связи) и *семантические отношения* (все остальные отношения).

4. ***Семантическая метрика***. Между информационными единицами устанавливают *отношения релевантности*, характеризующие ситуационную близость информационных единиц (ассоциативные связи). Отношение релевантности позволяет находить знания, близкие к уже найденным.

5. ***Активность***. Выполнение программ в информационных системах должно инициализироваться не командами, а состоянием информационной базы (фактами, связями между информационными единицами).

Перечисленные характеристики определяют разницу между *данными* и *знаниями*, а *базы данных* перерастают в *базы знаний.*

### *1.2.3. Модели представления знаний*

В ИС используются четыре основных типа моделей знаний.

1. ***Логические модели***. В основе логического способа представления знаний лежит идея описания знаний о предметной области в виде некоторого множества утверждений, выраженных в виде логических формул, и получение решения построением вывода в некоторой формальной системе. Логические модели описываются *формальной системой*, задаваемой четверкой вида: ***M = <T, P, A, B>***, где

*T – множество базовых элементов* различной природы.

*P – множество синтаксических правил*, с помощью которых из элементов *T* образуют *синтаксически правильные совокупности*. Например, из слов ограниченного словаря строятся синтаксически правильные фразы.

*A* – подмножество в множестве синтаксически правильных совокупностей. Элементы *A* называются *аксиомами*.

*B – множество правил вывода*. Применяя их к элементам *A*, можно получать новые синтаксически правильные совокупности, к которым снова можно применять правила из *B*. Таким образом формируется *множество выводимых* в данной формальной системе *совокупностей*.

Для знаний, входящих в БЗ, можно считать, что множество *A* образуют все информационные единицы, которые введены в базу знаний извне, а с помощью правил вывода из них выводятся новые *производные знания*. Другими словами, формальная система представляет собой генератор порождения новых знаний, образующих множество *выводимых* в данной системе *знаний*. Это свойство логических моделей делает их подходящими для использования в БЗ. Оно позволяет хранить в базе лишь те знания, которые образуют множество *A*, а все остальные получать по правилам вывода.

Основными задачами, решаемыми на логических моделях, являются следующие:

– установка или опровержение выводимости некоторой формулы (в общем случае эта задача алгоритмически неразрешима);

– доказательство полноты/неполноты некоторой формально логической системы, представленной множеством логических формул;

– определение следствий из заданной системы формул;

– доказательство эквивалентности двух формально-логических систем;

– доказательство теорем.

**Пример**

***Любить****(Саша, читать).* «Саша любит читать»

***Любить****(Саша, читать)* → ***Любить****(Саша, книги).* «Если Саша любит читать, то он любит и книги»

*Достоинства*:

1. Формальный аппарат вывода нового из известно.
2. Возможность контроля целостности.
3. Простая и ясная запись.

*Недостатки*:

1. Знания трудно структурировать.
2. Долгий вывод при большом числе правил (формул).
3. При большом числе правил (формул) их совокупность трудно обозрима.

2. ***Сетевые модели***. В основе моделей этого типа лежит конструкция, называемая семантическая сеть. *Семантическая сеть* – это граф, вершины которого соответствуют объектам или понятиям, а дуги определяют отношения между ними. Сетевые модели формально можно задать в виде:

$$H = \langle I, C_1, C_2, ..., C_n, G \rangle,$$

где $I$ – множество информационных единиц; $C_1$, $C_2$, ..., $C_n$ – множество типов связей между информационными единицами; $G$ – задает между информационными единицами, входящими в $I$, связи из заданного набора типов связей.

В зависимости от типов связей, используемых в модели, различают классифицирующие сети, функциональные сети и сценарии. В *классифицирующих сетях* используются отношения структуризации. Такие сети позволяют в БЗ вводить разные иерархические отношения между информационными единицами. *Функциональные сети* характеризуются наличием функциональных отношений (рис. 1.3). Их называют *вычислительными моделями*, т. к. они позволяют описывать процедуры определений одних информационных единиц через другие. В *сценариях* используются отношения типа «средство – результат». Если в сетевой модели допускаются связи различного типа, то ее обычно называют *семантической сетью*.

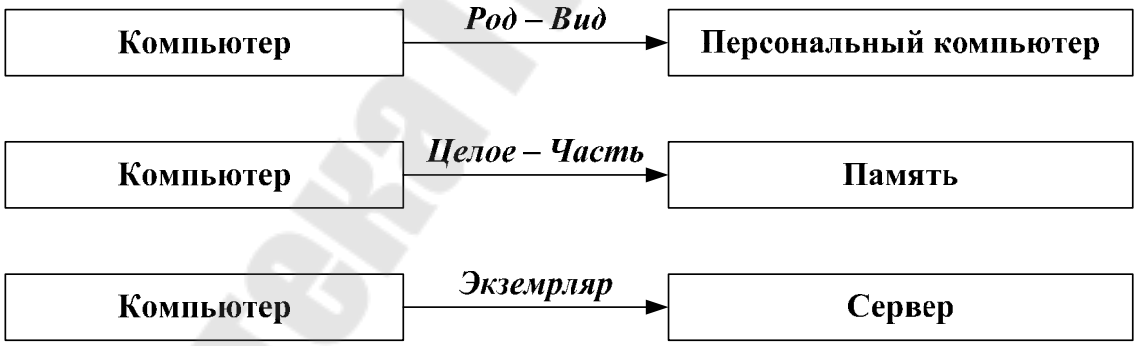



*Рис. 1.3*. Функциональные отношения

*Достоинства*:

1. Знания хорошо структурированы и понятны человеку.

*Недостатки*:

1. Долгий вывод при большом объеме сети.
2. При большом объеме сети она трудно обозрима.

3. ***Продукционные модели***. В моделях этого типа используются некоторые элементы логических и сетевых моделей. Из логических

моделей заимствована идея правил вывода, которые здесь называются *продукциями*, а из сетевых моделей – описание знаний в виде семантической сети. В результате применения правил вывода к фрагментам сетевого описания происходит трансформация семантической сети за счет смены ее фрагментов, наращивания сети и исключения из нее ненужных фрагментов. Таким образом, в продукционных моделях процедурная информация явно выделена и описывается иными средствами, чем декларативная информация. Вместо логического вывода, характерного для логических моделей, в продукционных моделях появляется вывод на знаниях.

В модели знаний на основе продукций знания представлены совокупностью правил в формате «*ЕСЛИ – ТО*». Продукции удачно моделируют человеческий способ рассуждений и находят широкое применение в экспертных системах.

**Пример**

**ЕСЛИ** «*Т>80 и Р>10КПа*» **ТО** «*открыть клапан*»

**ЕСЛИ** «*идет дождь*» **ТО** «асфальт мокрый»

*Достоинства*:

1. Простое описание знаний.
2. Независимость продукций и легкость модификации баз знаний.
3. Строгость, простота и изученность механизма вывода.

*Недостатки*:

1. Слабая структуризация баз знаний.
2. Неясность взаимных отношений продукций.
3. Не универсальность.

4. ***Фреймовые модели***. Фреймовая модель имеет жесткую структуру информационных единиц, которая называется *протофреймом*. В общем виде она выглядит следующим образом:

(***Имя фрейма***:

*Имя слота 1(значение слота 1)*

*Имя слота 2(значение слота 2)*

. . . . . .

*Имя слота К (значение слота К)).*

Значением *слота* может быть любая информация: числа или математические соотношения, тексты на естественном языке или программы, правила вывода или ссылки на другие слоты данного фрейма или других фреймов. При конкретизации фрейма ему и слотам присваиваются конкретные имена, происходит заполнение слотов. Таким образом, из протофреймов получаются *фреймы-экземпляры.* Связи между фреймами задаются значениями специального слота с именем «Связь».

**Пример**. Составить протофрейм, для заданной структуры.

| Фамилия | Группа | Курс |
|---|---|---|
| Попов | ИТ | 3 |
| Сидоров | ИТ | 2 |
| Иванов | ПЭ | 1 |
| Петров | ПС | 2 |

Протофрейм:
(**Список студентов**:
**Фамилия** (*Попов - Сидоров - Иванов - Петров*);
**Группа** (*ИТ - ИТ - ПЭ - ПС*);
**Курс** (*3 - 2 - 1 - 2*)).
Достоинства и недостатки те же, что и у сетевой модели.

# 2. ФУНКЦИОНАЛЬНОЕ И ЛОГИЧЕСКОЕ ПРОГРАММИРОВАНИЕ

## 2.1. Классификация языков и стилей программирования

Все языки программирования можно разделить на процедурные и декларативные языки [18]. Подавляющее большинство используемых в настоящее время языков программирования (Си, Паскаль, *Java* и т. п.) относится к процедурным языкам. Наиболее существенными классами декларативных языков являются функциональные (*Лисп*, *Лого*, *GNU Clisp*, *Python*, *Hope*, *Sisal*, *Haskell* и т. п. [14], [15], [19]) и логические (*Пролог*, *Плэнер*, *Конивер* и др.) языки программирования (рис. 2.1).

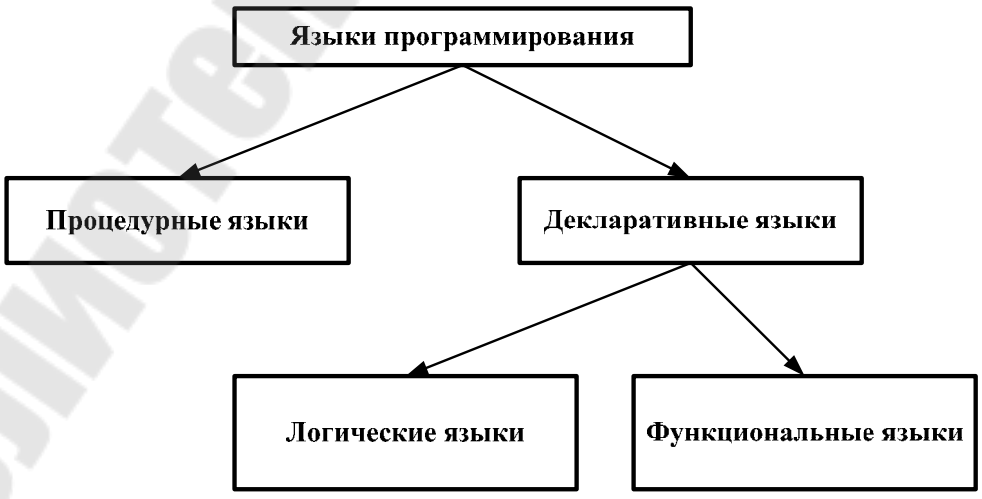



*Рис. 2.1*. Классификация языков программирования

На практике языки программирования не являются чисто процедурными, функциональными или логическими, а содержат в себе черты языков различных типов. На процедурном языке часто можно написать функциональную программу или ее часть и наоборот. Точнее было бы вместо типа языка говорить о стиле или методе программирования. Естественно, различные языки поддерживают разные стили в разной степени.

Программа на процедурном языке состоит из последовательности операторов и предложений, управляющих последовательностью их выполнения. Типичными операторами являются операторы присваивания и передачи управления, операторы ввода-вывода и специальные предложения для организации циклов. Из них можно составлять фрагменты программ и подпрограммы. В основе процедурного программирования лежит взятие значения какой-то переменной, а также совершение над ним действия и сохранения нового значения с помощью оператора присваивания, и так до тех пор, пока не будет получено (и, возможно, напечатано) желаемое окончательное значение.

***Логическое программирование*** – это один из подходов к информатике, при котором в качестве языка высокого уровня используется логика предикатов первого порядка в форме фраз Хорна [17]. Логика предикатов первого порядка – это универсальный абстрактный язык, предназначенный для представления знаний и для решения задач. Его можно рассматривать как общую теорию отношений. Логическое программирование базируется на подмножестве логики предикатов первого порядка, при этом оно одинаково широко с ней по сфере охвата. Логическое программирование дает возможность программисту описывать ситуацию при помощи формул логики предикатов, а затем для выполнения выводов из этих формул применить автоматический решатель задач (т. е. некоторую процедуру). При использовании языка логического программирования особое внимание уделяется описанию структуры прикладной задачи, а не выработке предписаний компьютеру, что ему следует делать [1], [6]. Другие понятия информатики из таких областей, как теория реляционных БД, программная инженерия и представление знаний, также можно описать (и, следовательно, реализовать) с помощью логических программ.

***Функциональное программирование*** предполагает, что программа состоит из совокупности определений функций [9]. Функции, в свою очередь, представляют собой вызовы других функций и предложений,

управляющих последовательностью вызовов. Вычисления начинаются с вызова некоторой функции, которая в свою очередь вызывает функции, входящие в ее определение и так далее в соответствии с иерархией определений и структурой условных предложений. Функции часто либо прямо, либо опосредованно вызывают сами себя [18].

Каждый вызов возвращает некоторое значение в вызывавшую его функцию, вычисление которой после этого продолжается; этот процесс повторяется до тех пор, пока запустившая вычисления функция не вернет конечный результат пользователю. «Чистое» функциональное программирование не признает присваиваний и передач управления. Разветвление вычислений основано на механизме обработки аргументов условного предложения. Повторные вычисления осуществляются через рекурсию, являющуюся основным средством функционального программирования.

## 2.2. Функциональное программирование

### *2.2.1. История развития функционального программирования*

Язык Лисп был разработан в Стэнфорде под руководством Джона МакКарти (John McCarthy) в 1958 г. Он является вторым по возрасту (первый – Фортран) языком программирования [9]. По первоначальному замыслу он должен был включать наряду со всеми возможностями Фортрана средства работы с матрицами, указателями и структурами из указателей и т. п. Но для такого проекта не хватило средств. Окончательно сформированные принципы, положенные в основу языка Лисп, следующие: использование единого спискового представления для программ и данных; применение выражений для определения функций; скобочный синтаксис языка. Название языка происходит от *List processing* (обработка списков). Лисп представляет собой язык функционального программирования и основан на алгебре списочных структур, лямбда-исчислении и теории рекурсивных функций. Существует более трехсот диалектов Лиспа и родственных ему языков (*Interlisp*, *muLisp*, *CommonLisp*, *MicroLisp*, *MuLisp*, *Sail*, *Hope*, *Miranda*, *Scheem*, *ML*, *GNU Clisp*, *Elisp*, *xLisp*, *Vlisp*, *AutoLisp* [14], *Haskell* [12], *Python*, *CMUCL* и т. д.), среди которых можно выделить *Common Lisp* (1984) [19], который получил наибольшее распространение. Этот диалект является первым объектноориентированным языком, стандартизированным *ANSI* (*ANSI standard X*3.226 1994).

По разнообразию типов данных Лисп превосходит любой другой язык, причем типы привязываются к значению переменной, а не к имени, поэтому легко создавать разнотипные списки и массивы.

Определение функций и их вычислений в Лиспе основано на лямбда-исчислении (*lambda calculus*) Черча [2]. Функции в Лиспе могут передаваться как параметры, модифицироваться и возвращаться как значения. Кроме того, в Лиспе компилятор можно вызывать из исполняемой программы «на лету», т. е. для модификации и рекомпиляции отдельной функции или класса не требуется остановка системы.

Таким образом, можно выделить следующие отличия Лиспа:

1. Лисп – функциональный язык рекурсивного программирования, ориентированный на решение задач ИИ. Функциональное программирование основывается на том, что в результате каждого действия возникает значение. Значения становятся аргументами следующих действий, и конечный результат всей задачи выдается пользователю. Программы строятся из логически расчлененных определений функций. Лисп наиболее эффективен при проведении рекурсивных вычислений.

2. Ранние реализации языка были интерпретаторами и особой скоростью не отличались. В настоящее время все коммерческие и некоторые бесплатные реализации имеют качественные оптимизирующие компиляторы.

3. Лисп – это язык с динамической типизацией. В Лиспе типы привязываются к значению переменной, а не к ее имени. Это позволяет легко создавать разнотипные списки и массивы.

4. Функции в Лиспе могут передаваться как параметры, модифицироваться и возвращаться как значения. Кроме того, компилятор можно вызывать из исполняемой программы «на лету».

5. Объектная система Лиспа (*CLOS – Common Lisp Object System*) по своим возможностям значительно совершеннее того, что предлагает *C++*. Метаобъектный протокол позволяет кардинально изменять свойства объектной системы (например, порядок вызова методов при наследовании). В *CLOS* методы могут привязываться сразу к нескольким различным классам.

6. Определение функций и их вычислений в Лиспе основано на лямбда-исчислении (*lambda calculus*) Черча.

7. Лисп предназначен для решения задач ИИ – экспертные системы, системы поддержки принятия решений, синтез речи, распознавание образов, компьютерное зрение и т. п.

На Лиспе написаны программа управления орбитальным телескопом *Hubble*, программное обеспечение контроля бельгийских атомных станций, система электронной коммерции *Yahoo! Store*, система автоматического устранения неполадок и реконфигурации космических зондов серии *Deep Space*, система планирования воздушных атак министерства обороны США, АСУ сборочных конвейеров *Ford Motors* и *General Motors* и т. д. Причина заключается в высокой надежности языка и в поддержке различными платформами. Кроме функционального программирования в Лиспе можно использовать программирование, основанное на обычном последовательном исполнении операторов с присваиваниями, передачами управления и специальными операторами цикла.

### *2.2.2. Представление данных в Лиспе*

Данные в Лиспе представляются атомами, списками, консами, символьными выражениями. Взаимосвязь понятий представлена на рис. 2.2.

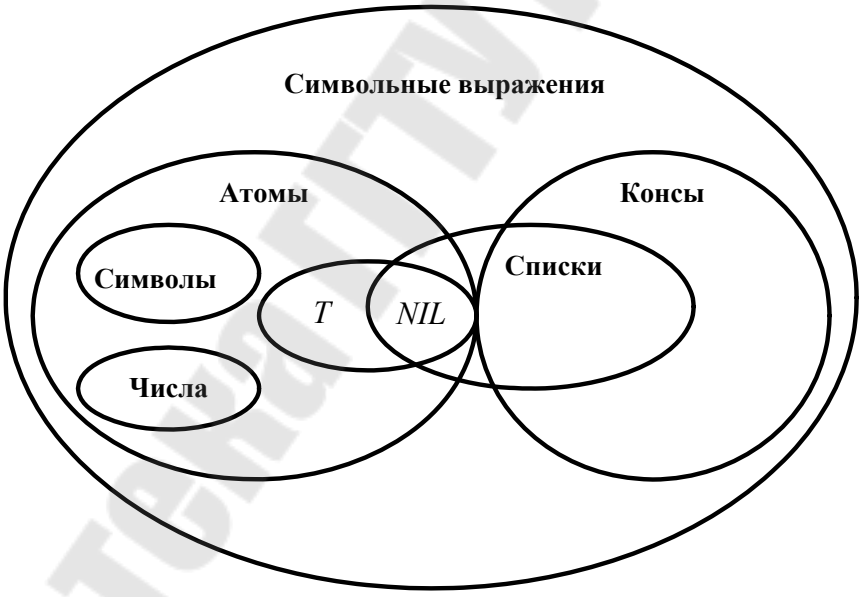



*Рис. 2.2*. Символьные выражения Лиспа

***Атомы*** – простейшие объекты Лиспа, делятся на символьные и числовые. ***Символьный атом*** – это последовательность букв, цифр и, возможно, специальных символов, например *X*1, *IT*–41, *ABCD*. ***Числовой атом*** – это последовательность цифр, например 2, –75, 355. В числовом атоме могут присутствовать символы «. + – · /», например –11, +55.473, 2/5. Числовые атомы интерпретируются как константы,

символьные – как константы и как переменные. Атом ***T*** обозначает логическое значение «истина», атом ***NIL*** – логическое значение «ложь» или пустой список.

Последовательность элементов, разделенных пробелами и заключенных в круглые скобки, является списком. Элементами списка могут быть любые объекты – атомы, списки, консы, например (1 *a* 2 *b* 3 *c*), ((*x*1 0) (*x*2 1)), ((*y* . *blue*) (*z* . *yellow*)). Пустой список не содержит элементов и обозначается ( ) или *NIL*. Таким образом, ***список*** – это многоуровневая или иерархическая структура данных, в которой открывающие и закрывающие скобки находятся в строгом соответствии. Например, приведенные ниже выражения являются правильно составленными списками:

(+ 2 3) – список из трех элементов;

(((((первый) 2) третий) 4) 5) – список из двух элементов.

Список, в котором нет ни одного элемента, называется *пустым списком*. Пустой список – это не то же самое, что «ничего». Он выполняет ту же роль, что и нуль в арифметике. ***NIL*** может быть, например, элементом других списков:

*NIL* – то же, что и пустой список или ( );

(*NIL*) – список, состоящий из атома *NIL*;

(( )) – то же, что и (*NIL*);

((( ))) – то же, что и ((*NIL*));

(*NIL* ( ) ) – список из двух пустых списков.

Пара элементов, разделенных точкой и заключенных в круглые скобки, называется *консом*, или *точечной парой*. Список (*e*1 *e*2 … *en*) может быть представлен суперпозицией консов – (*e*1 . (*e*2 . (… (*en* . *NIL*) ))).

Символьным выражением в Лиспе является один из следующих объектов: атом, список (*s*1…*sn*) или конс (*s*1.*s*2), где *s*1, *s*2, …, *sn* – символьные выражения, например (*s*3 (2 . *L*) . (*a c*)) является символьным выражением.

Сложные структуры в Лиспе представляются с использованием списочных ячеек (рис. 2.3, *а*). Так, точечная пара (*a.b*) представляется одной списочной ячейкой (рис. 2.3, *б*), в то время как список (*a b*) представляется двумя ячейками (рис. 2.3, *в*).

Первый элемент списка называется *головой* (*head*), а остаток списка, т. е. список без первого его элемента, называется *хвостом списка* (*tail*). Левая часть списочной ячейки указывает на голову списка (или левую часть конса), а правая – на хвост списка (или правую

часть конса). При помощи селекторов *CAR* и *CDR* можно выделить из списка его голову и хвост.

Функция *CAR* [объект] возвращает *car*-элемент <объекта> (точнее, объект данных, на который указывает *car*-элемент <объекта>). Интерпретация *car*-элемента объекта зависит от того, чем именно является объект. Если объект – атом, то его *car*-элемент есть текущее значение атома. Если объект – список, его *car*-элемент есть первый элемент списка (голова списка). Если объект – бинарное дерево, его *car*-элемент есть левая ветвь дерева:

(*CAR* '(*A B C D E*)) —> *A*

(*CAR* '((*A* . *B*) . *C*)) —> (*A* . *B*).

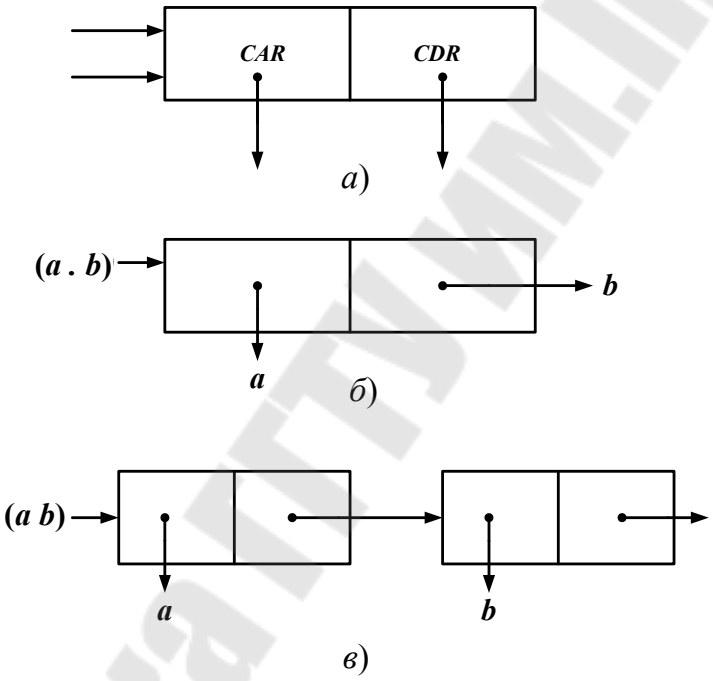



*Рис. 2.3*. Представление объектов Лиспа:
*а* – списочная ячейка; *б* – точечная пара (конс); *в* – список

Функция *CDR* [объект] возвращает *cdr*-элемент <объекта> (объект данных, на который указывает *cdr*-элемент <объекта>). Если объект – символ, его *cdr*-элемент есть список свойств символа. Если объект – число, его *cdr*-элемент указывает признак и тип числа. Если объект – список, его *cdr*-элемент есть остаток списка (т. е. все, кроме последнего, элементы списка). Если объект – бинарное дерево, его *cdr*-элемент есть правая ветвь дерева:

(*CDR* '(*A B C*)) —> (*B C*)

(*CDR* '((*A.B*) . *C*)) —> *C*.

Функция *CONS* [объект1, объект2] возвращает точечную пару, у которой *car*-элемент указывает на <объект1>, а *cdr*-элемент – на <объект2>. Если <объект2> – список, *CONS* создает список, первым элементом которого является <объект1>, а остатком – <объект2>. Если <объект2> – атом или бинарное дерево, *CONS* создает дерево, у которого левая ветвь есть <объект1>, а правая ветвь – <объект2>:

(*CONS* '*A* '(*B C D*)) —> (*A B C D*)

(*CONS* '*A* '*B*) —> (*A* . *B*).

Функция *EQ* [объект1, объект2] выполняет роль предиката и проверяет на идентичность два своих аргумента (т. е. они указывают на одинаковые объекты данных в памяти). В качестве результата возвращается ***T*** или ***NIL***. Лисп использует достаточно большое количество предикатов: (*NULL S*) – список пуст; (*ZEROP N*) – *N*=0; (*ATOM A*) – *A* является атомом и т. п.

Предложение *COND* предназначено для организации ветвления:

(*COND*

((*predicate*$_1$) (*form*$_1$))

((*predicate*$_2$) (*form*$_{21}$) (*form*$_{22}$) … (*form*$_{2M}$))

(*predicate*3)

…

((*predicate*$_N$) (*form*$_N$))

)

При выполнении предложения *COND* последовательно вычисляются значения предикатов, обозначенных как *predicate*. Если предикат возвращает значение ***T*** (более корректно, не ***NIL***), тогда вычисляется значение вычислимой формы *form* и полученное значение возвращается в качестве значения всего предложения *COND*. Другими словами, идет последовательный перебор предикатов до тех пор, пока не встретится предикат, который будет не равен ***NIL***.

Для некоторого предиката может отсутствовать вычислимая форма. В таком случае предложение *COND* возвратит значение самого предиката. Для некоторого предиката вычислимых форм может быть несколько. В таком случае формы будут вычислены последовательно и значение последней формы будет возвращено как значение всего предложения *COND*.

В Лиспе формы представления программы и обрабатываемых ею данных одинаковы и представляются списочной структурой. Списки, представляющие программы и данные, состоят из списочных ячеек, расположение и порядок которых в памяти несущественный.

Структура списка определяется логически на основе имен символов и указателей. Добавление новых элементов в список или удаление из списка может производиться без переноса списка в другие ячейки памяти. Резервирование и освобождение могут в зависимости от потребности осуществляться динамически, ячейка за ячейкой.

### *2.2.3. Лямбда–исчисление*

Определение функций и их вычисление в Лиспе основаны на лямбда-исчислении Черча. ***Лямбда-выражение*** – это определение вычислений и параметров функции в чистом виде, т. е. без фактических параметров или аргументов, которое имеет вид:

(*LAMBDA* ($x_1$ $x_2$…$x_n$) $f_n$).

Символ *LAMBDA* обозначает, что мы имеем дело с определением функции. Символы $x_i$ являются формальными параметрами определения, которые именуют аргументы в описывающем вычисления теле функции $f_n$. Входящий в состав формы список, образованный из параметров, называется ***лямбда-списком***.

Телом функции является произвольная форма, значение которой может вычислить интерпретатор Лиспа, например, константа, или связанный со значением символ, или композиция из вызовов функций.

В математике и обычных языках программирования вызов функции записывается, как правило, в так называемой префиксной записи (нотации), в которой имя функции стоит перед скобками, окружающими аргументы:

*f*(*x*), *g*(*x*,*y*), *h*(*x*, *g*(*y*,*z*)).

Определение функции в языке Лисп позволяет задать произвольное вычислимое отображение. Например, функцию, вычисляющую сумму квадратов, можно определить с помощью сложения и умножения:

(*defun сумма_квадратов* (*x y*)

(+ (* *x x*) (* *y y*)))

имя – *сумма_квадратов*; аргументы – *x, y*; значение – *x*x+y*y*.

Используя лямбда-выражение, данную функцию можно записать:

(*LAMBDA* (*x y*) (+ (* *x x*) (* *y y*))),

где *x* и *y* – формальные параметры, представляющие произвольные числа. Формальность параметров означает, что их можно заменить на любые другие символы и это не отразится на вычислениях, определяемых функцией. Именно это и скрывается за лямбда-нотацией.

Вызов функции обозначает запуск или применение определения функции к конкретным значениям аргументов. Например, применим функцию *сумма квадратов* к фактическим аргументам $x = 2$ и $y = 3$:

(*сумма_квадратов* 3 4) $\rightarrow$ 25.

Используемое в Лиспе функциональное программирование основывается на том, что в результате каждого действия возникает значение. Значения становятся аргументами следующих действий, и конечный результат всей задачи выдается пользователю. Программы строятся из логически расчлененных определений функций. Кроме функционального программирования в Лиспе можно использовать программирование, основанное на обычном последовательном исполнении операторов с присваиваниями, передачами управления и специальными операторами цикла.

В Лиспе существуют специальные функции – *предикаты*, которые проверяют наличие у символьного выражения некоторого свойства. Другими словами, предикат – это функция, которая определяет, обладает ли аргумент определенным свойством, и возвращает в качестве значения «ложь» (или ***NIL***) в случае, если аргумент не обладает этим свойством, и «истина» (или ***T***) – в противном случае.

## 2.3. Понятие рекурсии

### *2.3.1. Рекурсивный подход к вычислениям*

Функциональная программа состоит из совокупности определений функций. Функции, в свою очередь, представляют собой вызовы других функций и предложений, управляющих последовательностью вызовов. Вычисления начинаются с вызова некоторой функции, которая в свою очередь, вызывает функции, входящие в ее определение и так далее в соответствии с иерархией определений и структурой условных предложений. Функции часто либо прямо, либо опосредованно вызывают сами себя.

Каждый вызов возвращает некоторое значение в вызвавшую его функцию, вычисление которой после этого продолжается; этот процесс повторяется до тех пор, пока запустившая вычисления функция не вернет конечный результат пользователю. «Чистое» функциональное программирование не признает присваиваний и передач управления. Разветвление вычислений основано на механизме обработки аргументов условного предложения. Повторные вычисления осуществляются через рекурсию, являющуюся основным средством функционального программирования.

Программа (подпрограмма) является рекурсивной, если в процессе своей работы она прямо или косвенно вызывает сама себя. Кроме того, она всегда содержит, по крайней мере, одну терминальную ветвь (условие окончания вычислений) и хотя бы одну рекурсивную ветвь (условие продолжения вычислений).

**Пример**

Определим функцию, которая вычисляет факториал целого числа *n*. Для этого необходимо перемножить все числа от 1 до *n*. Из математики известно, что $n! = n(n-1)!$, т. е. факториал *n* равен произведению *n* и факториала от $(n-1)$. Запишем определение функции (считая, что 0!=1):

```
(defun fact (n)
    (cond
        ((zerop n) 1)                ;прекращение вычислений
        (t (* n (fact (– n 1))))     ;продолжение вычислений
))
```

В данном примере функция *fact* рекурсивно вызывает сама себя с аргументом, который на единицу меньше значения предыдущего вызова. Рекурсивный вызов продолжается до тех пор, пока значение аргумента не станет равно нулю:

```
(fact 3)     (3*fact(2))                    3*2=6
                (2*fact(1))                 2*1=2
                   (1*fact(0))              1*1=1
```

Таким образом, получили $1 \cdot 2 \cdot 3 = 6$.

В Лиспе рекурсия основана на математической теории рекурсивных функций. Рекурсия хорошо подходит для работы со списками, т. к. сами списки могут состоять из подсписков, т. е. иметь рекурсивное строение. Для обработки рекурсивных структур совершенно естественно использование рекурсивных процедур.

Определение языка программирования обычно начинают с синтаксических формул, называемых *БНФ* (формулы Бекуса–Наура). Определение таких формул сводится к следующим положениям:

– язык характеризуется набором определяемых понятий;

– каждому понятию соответствует набор вариантов синтаксических формул;

– каждый вариант – это последовательность элементов;

– элемент – терминальный символ или синтаксическое понятие.

Синтаксис данных в Лиспе сводится к правилам представления атомов и *S*-выражений.

*<атом>* ::= *<буква>* *<конец_атома>*

*<конец_атома>* ::= *<пусто>*

| *<буква>* *<конец_атома>*

| *<цифра>* *<конец_атома>*

В Лиспе атомы – это мельчайшие частицы. Их разложение по литерам обычно не имеет смысла.

*<S-выражение>* ::= *<атом>*

| (*<S-выражение>* . *<S-выражение>*) ; точечная пара (конс)

| (*<S-выражение>* *<S-выражение>* … ); список

По этому правилу *S-выражения* – это атомы, точечные пары или списки из *S-выражений*.

Базовая система кодирования данных – *точечная нотация*. Однако для человека представление данных в виде списков гораздо удобнее. В Лиспе достаточно просто перейти от одной формы представления к другой:

() ↔ ***NIL***

(*S* . ***NIL***) ↔ (*S*)

($S_1$ . ( ... ($S_k$ . ***NIL***) ... )) = ($S_1$ ... $S_k$)

Такая единая структура данных оказалась вполне достаточной для представления любых сложных программ в виде двоичных деревьев. Дальнейшее определение языка Лиспа можно рассматривать как восходящий процесс генерации семантического каркаса, по ключевым позициям которого распределены семантические действия по обработке программ.

Таким образом, списки можно определить при помощи следующих правил Бэкуса-Наура:

*список* ↔ ***NIL*** ; список либо пуст, либо это

*список* ↔ (*голова* . *хвост*) ; точечная пара, хвост которой является списком (рекурсия «в ширину»)

*голова* ↔ *атом*

*голова* ↔ *список* ; рекурсия «в глубину».

Рекурсия есть как в определении головы, так и в определении хвоста списка. Заметим, что приведенное выше определение напрямую отражает определения функций, работающих со списками, которые могут обрабатывать рекурсивным вызовом голову списка, т. е. «в глубину» (в направлении *CAR*), и хвост списка, т. е. «в ширину» (в направлении *CDR*).

### *2.3.2. Виды рекурсии*

Функция является рекурсивной, если в ее определении содержится вызов самой этой функции. Будем говорить о рекурсии *по значению*, когда вызов является выражением, определяющим результат функции (например, функция *fact*, рассмотренная выше). Если же в качестве результата функции возвращается значение некоторой другой функции и рекурсивный вызов участвует в вычислении аргументов этой функции, то будем использовать рекурсии *по аргументам*.

Рассмотрим пример копирования списков:

```
(defun copy (S)
        (cond
                ((null S) nil)
                (t (cons (car S) (copy (cdr S)))) ))
```

В данном случае имеем рекурсию по аргументу, т. к. рекурсивный вызов стоит на месте второго аргумента функции *cons*.

Аргументом рекурсивного вызова может быть вновь рекурсивный вызов, и таких вызовов может быть много. Рекурсия *простая*, если вызов функции встречается в некоторой ветви программы только один раз (функции *fact* и *copy*). Простой рекурсии в процедурном программировании соответствует обыкновенный цикл. Однако в определении функции (или в определениях вызывающих друг друга функций) рекурсия может принимать различные формы, а именно:

1. *Параллельная рекурсия*. Тело определения функции *f* содержит вызов некоторой функции *g*, несколько аргументов которой являются рекурсивными вызовами функции *f*:

```
(defun f ...
... ( g ... (f ... ) ... ( f ... ) ... )
…).
```

2. *Взаимная рекурсия*. В определении функции *f* вызывается некоторая функция *g*, которая содержит вызов функции *f*:

```
(defun f ...
... ( g ... ) ... )
(defun g ...
... ( f ... ) ... )
```

3. *Рекурсия более высокого порядка*. В данном случае аргументом рекурсивного вызова является рекурсивный вызов:

```
(defun f ...
... ( f ... ( f ...) ... )
… )
```

Рассмотрим вышеперечисленные виды рекурсии более подробно.

Рекурсию называют *параллельной*, если она встречается одновременно в нескольких аргументах функции. Так выглядят повторяющиеся вычисления, соответствующие следующим друг за другом (текстуально) циклам в операторном программировании.

Рассмотрим в качестве примера копирование списочной структуры на всех уровнях. Будем рассматривать список как бинарное дерево (*binary tree*), у которого левое поддерево соответствует голове списка, а правое поддерево – хвосту. Возможно три варианта дерева:

```
–дерево > nil                          ; пустое дерево
–дерево > атом                         ; лист дерева
–дерево > (дерево . дерево)            ; дерево точечная – пара
(defun copy-tree (S)
      (cond
      ((null S ) nil)
      ((atom S) S)
      (t (cons (copy-tree (car S )) (copy-tree (cdr S )) )) ))
                  копия головы       копия хвоста
```

Здесь рекурсия применяется как к голове, так и к хвосту списка. Рекурсивные вызовы представляют собой два аргумента одной функции *cons*, т. е. имеет место параллельная рекурсия. Однако параллельность является лишь текстуальной или логической, т. к. вычисление ветвей производится последовательно.

Рекурсия является *взаимной* между двумя или более функциями, если они вызывают друг друга. Для примера можно представить ранее определенную нами функцию обращения списков на всех уровнях в виде двух взаимно рекурсивных функций следующим образом:

```
(defun ReverseM (S)
      (cond
            ((atom S) S)
            (t (Exch S nil)) ))
(defun Exch (S rez)
      (cond
            ((null S) rez)
            (t (Exch (cdr S) (cons (ReverseM (car S)) rez) )) ))
```

Функция *Exch* используется в качестве вспомогательной функции. В процессе построения обращенного списка она заботится и о том, чтобы возможные подсписки были обращены. Она делает это не сама, а передает эту работу специализирующейся на этом функции

*ReverseM*. Результат получен взаимной рекурсией. Глубина и структура рекурсии зависят от строения списка *S*. Кроме участия во взаимной рекурсии функция *Exch* рекурсивна и сама по себе.

С помощью рекурсии легко работать с динамическими, заранее не определенными целиком, но достаточно регулярными структурами, такими как списки произвольной длины и глубины вложения.

Используя все более мощные виды рекурсии, можно записывать относительно лаконичными средствами и более сложные вычисления. Одновременно с этим, поскольку определения довольно абстрактны, растет сложность программирования и понимания программы.

Рассмотрим теперь программирование вложенных циклов в такой форме, при которой в определении функции рекурсивный вызов является аргументом вызова этой же самой функции. В такой рекурсии можно выделить различные порядки в соответствии с тем, на каком уровне рекурсии находится вызов. Такую форму рекурсии будем называть рекурсией *более высокого порядка*. Функции, которые были определены выше, являются функциями с рекурсией нулевого порядка. В качестве классического примера рекурсии более высокого порядка приведем определение функции *rev*, использующей лишь базовые функции и рекурсию:

```
(defun rev (S)
  (cond
    ((null S) S)
    ((null (cdr S)) S)
    (t (cons (car (rev (cdr S))) (rev (cons (car S) (rev (cdr (rev (cdr S))))))))
))
```

В определении использована рекурсия второго порядка. Вычисления, представленные этим определением, понять труднее, чем прежние. Сложная рекурсия усложняет и вычисления. В этом случае невозможно вновь воспользоваться полученными ранее результатами, поэтому одни и те же результаты приходится вычислять снова и снова. Обращение списка из пяти элементов функцией *rev* требует 149 вызовов. Десятиэлементный список требует уже 74 409 вызовов и заметное время для вычисления! Как правило, многократных вычислений можно избежать, разбив определение на несколько частей и используя подходящие параметры для сохранения и передачи промежуточных результатов. Обычно в практическом программировании формы рекурсии более высокого порядка не используются.

# 3. ЛОГИЧЕСКОЕ ПРОГРАММИРОВАНИЕ

## 3.1. Основы языка Пролог

В Прологе (*Prolog – PROgramming LOGic*) решение задачи реализуется логическим выводом из ранее известных положений. Программа на Прологе не является последовательностью действий, а представляет собой набор фактов и правил, обеспечивающих получение выводов на их основе. Поэтому Пролог считается *декларативным языком программирования.*

Пролог базируется на предложениях Хорна, являющихся подмножеством формальной системы, называемой *логикой предикатов.* Механизм вывода основан на сопоставлении образцов. С помощью подбора ответов на запросы Пролог извлекает хранящуюся (известную) информацию, заданную в программе.

Особенность Пролога – дополнение к логическому поиску ответов на поставленные вопросы, он находит все возможные альтернативные решения. Вместо обычной работы от начала программы до ее конца Пролог может возвращаться назад и просматривать более одного «пути» при решении всех составляющих задачу частей.

Программист на Прологе описывает объекты (*objects*) и отношения (*relations*), а затем описывает правила (*rules*), при которых эти отношения являются истинными.

Обычно программа на Пролог состоит из следующих разделов:

*описание параметров компиляции*
*CONSTANTS*
*DOMAINS*
*FACTS или DATABASE*
*PREDICATES*
*CLAUSES*
*GOAL*

В программе не обязательно должны быть все эти разделы. Как правило, программа содержит, по меньшей мере, разделы *PREDICATES, CLAUSES, GOAL*. В программе может быть несколько разделов описаний *DOMAINS, PREDICATES, DATABASE* и *CLAUSES*. Однако разделов *GOAL* не может быть в программе более одного. Порядок разделов может быть произвольным, но при этом *константы*, *домены* и предикаты должны быть определены до их использования. Рассмотрим разделы программы более подробно.

## 3.2. Раздел констант и доменов (*CONSTANTS, DOMAINS*)

Объявление константы имеет вид:

```
<имя константы>=<значение>
```

Каждое определение константы должно размещаться в отдельной строке.

**Пример 3.1.** Задание числовых и символьных констант:

```
CONSTANTS
pi=3.14
data_path="d:\\data"
```

В разделе *DOMAINS* объявляются невстроенные домены, которые называют доменами пользователя. Домены позволяют задавать имена различным типам данных. Возможны следующие варианты объявления доменов.

1. ***Использование встроенных доменов.*** Объявление домена имеет следующий вид:

```
<имя_домена1>,.. <имя_доменаN>=<определение домена>
```

Важно, что описанные домены *имя_домена1 ... имя_доменаN* считаются **различными**, несмотря на объявление от одного домена. Встроенные домены представлены в табл. 3.1.

*Таблица 3.1*

**Основные стандартные домены**

| Домен | Описание и реализация |
|---|---|
| *byte* | Целое число (0…255) |
| *word* | Целое число (0…65 535) |
| *integer* | Целое число (–32768...32767) |
| *long* | Целое число (–2 147 483 648…2 147 483 647) |
| *real* | Вещественное число (±1e–307...±1e308) |
| *char* | Символ, заключенный в одинарные кавычки, реализуемый как byte |
| *string* | Последовательность символов, заключенная в двойные кавычки, реализуемая как указатель |
| *symbol* | Символическая константа. Реализуется как указатель на вход в таблице идентификаторов (хэш), хранящей строки идентификаторов. Кавычки для описания не требуются |
| *file* | Файловый домен |

2. ***Составные, или структурные домены*** **(структуры)**. Составные объекты данных позволяют обрабатывать несколько частей информации как единый элемент. Общий вид ***структурного домена***:

```
имя_домена = functor(домен1, домен2,..., доменN)
```

***Функтор*** – является идентификатором составных данных. При объявлении составного домена допускается совпадения имен составного домена и функтора.

**Пример 3.2.** Описание простых и структурных доменов. Структурный домен для обозначения даты – месяц, день, год.

```
DOMAINS
   name, sex = symbol
   age = integer
   date = date(string, unsigned, unsigned)
```

Пример использования:

```
Name=Ivan, Sex=M, Age=20
D = date("Февраль",23,2011)
```

Имя функтора позволяет различать объекты, объявленные составным доменом. Общий вид задания ***составного домена***:

```
domain = domain1; domain2;... domainN
```

где *domain1 … domainN* – произвольные (но отличные) функторы;

; – используется для задания альтернатив описания.

Каждая компонента структуры, в свою очередь, может быть структурой.

**Пример 3.3.** Зададим составной объект для описания треугольника в различных системах координат:

```
DOMAINS
   point = p2(integer, integer);
           p3(integer, integer, integer)
PREDICATES
   triangle(point, point, point)
```

В данном примере аргументы (*point*) предиката, описывающего треугольник *triangle*, автоматически будут приводиться к различным доменам *p*2 или *p*3.

## 3.3. Раздел предикатов (*PREDICATES*)

Раздел используется для объявления собственных (не встроенных) предикатов. Предикаты задают факты и правила. Объявление предиката имеет следующий вид:

```
<режим_детерминизма> имя (домен_определения)
```

К имени предиката предъявляются стандартные требования. Регистр букв не имеет значения, однако не следует использовать **заглавные буквы в качестве первой буквы имени предиката**. Аргументы предикатов должны принадлежать доменам, известным Пролог. Домены могут быть либо стандартными, либо объявленными в разделе доменов (пользовательские).

Режим детерминизма предиката сообщает компилятору может ли согласование предиката быть неудачным и могут ли остаться точки отката после успешного согласования предиката. Возможно несколько значений режимов детерминизма предикатов:

1) *determ* – согласование предиката может быть неудачным; после успешного согласования **точки отката не остаются** (режим по умолчанию);

2) *nondeterm* – согласование предиката может быть неудачным, после успешного согласования **могут остаться точки отката**;

3) *procedure* – согласование предиката всегда успешно, причем точек отката остаться не может.

*Visual Prolog* содержит большой набор встроенных предикатов. Все встроенные предикаты являются детерминированными.

**Пример 3.4.** Объявления двух предикатов с различными режимами детерминизма.

```
PREDICATES
     first(integer, integer)
     nondeterm second (char)
```

Один предикат может иметь несколько описаний. Это используется, когда необходимо, чтобы предикат работал с различным числом аргументов, различной природы. ***Арность предиката*** – это количество его аргументов. Пролог позволяет задавать предикаты с одним и тем же именем, но с различной арностью и доменами. **Предикаты с различные арностью должны группироваться вместе**. Кроме этого предикаты с различной арностью обрабатываются как абсолютно разные предикаты.

**Пример 3.5.** Объявление предикатов различной арности.

```
DOMAINS
   person = symbol
PREDICATES
   father(person)  % Этот человек отец
   father(person, person)  % Один человек является
отцом другому человеку
```

При объявлении предикатов можно указывать их шаблоны:

```
имя_предиката (домен) – (шаблон_1)…(шаблон_N)
```

Шаблон предиката указывает компилятору на то, какие аргументы предиката могут быть связанными (*i* – от *input*), а какие свободными (*o* – от *output*) на момент согласования предиката.

**Пример 3.6.** Объявляется предикат sum с тремя шаблонами. Для того чтобы компилятор смог «разобраться», к какому из трех правил ему следует обратиться, используются два встроенных предиката *free* (аргумент связан) и *bound* (аргумент свободен).

```
PREDICATES
nondeterm sum(integer, integer, integer) - (i, i, o) (o, i, i) (i, o, i)
CLAUSES
  sum(X,Y,Z):- bound(X), bound(Y), free(Z), Z=X+Y.
  sum(X,Y,Z):- free(X), bound(Y), bound(Z), X=Z-Y.
  sum(X,Y,Z):- bound(X), free(Y), bound(Z), Y=Z-X.
GOAL
X1=5, Y1=4, sum(X1, Y1, Z1).    % на экране 9
X2=5, Z2=1, sum(X2, Y2, Z2).    % на экране -4
```

## 3.4. Раздел предложений (*CLAUSES*)

Раздел предложений считается основным и содержит все факты и правила, составляющие программу, поэтому программу на Пролог, иногда называют БЗ, состоящей из предложений (или утверждений). Предложения бывают двух видов: факты, правила. Общий вид предложения имеет вид:

```
A:- B1,... , Bn,
```

где – *A* называется заголовком или головой предложения, а *B*1,..., *Bn* – телом.

***Заголовок*** – это факт, который истинен, если истинно тело. ***Тело*** – это ряд условий, которые проверяются для доказательства. ***Факты*** – это отношения или свойства, о которых известно, что они истинны. ***Факт*** является основой для логического вывода и состоит из имени отношения и объектов, заключенных в круглые скобки. Факт констатирует, что между объектами выполнено некоторое отношение. Можно считать, что факт – это предложение, у которого тело пустое. Факт завершается точкой (**.**). Факт представляет собой безусловно истинное утверждение.

**Пример 3.7.** Предложение на естественном языке: «*Билл любит собак*.» (Bill likes dogs) в синтаксисе Пролог будет выглядеть:

```
likes (bill, dogs).
```

Факты могут выражать и свойства. Предложение на естественном языке: «Kermit is green» (Лягушка зеленая) в синтаксисе Пролог будет выглядеть:

```
green (kermit).
```

***Правила*** – это связанные отношения, позволяющие реализовать логический вывод одной информации из другой. ***Правило*** – это предложение, истинность которого зависит от истинности одного или нескольких предложений. Тело правила состоит из одной или более подцелей и заканчивается точкой. Подцели разделяются запятыми, которые задают конъюнкцию подцелей. Общий вид:

```
заголовок:- <Подцель>, <Подцель>,..., <Подцель>.
```

Символ **:–** можно читать как «если». Все предложения для каждого конкретного предиката в разделе CLAUSES должны располагаться **вместе**. Последовательность предложений, описывающих один предикат, называется *процедурой*. Можно рассматривать правило и как процедуру.

**Пример 3.8.** Задание правил:

Таня любит то же, что и Вася:

```
likes (Таня, Something):- likes (Вася, Something).
```

Таня любит все зеленое:

```
likes (Таня, Something):- green (Something).
```

## 3.5. Раздел фактов и цели (*DATABASE, GOAL*)

Программа на Прологе представляет собой набор фактов и правил. Часто в процессе работы программы возникает необходимо модифицировать (изменить, удалить или добавить) некоторые факты. В этом случае факты рассматриваются как *динамическая*, или *внутренняя*, БД, которая при выполнении программы может изменяться. Для объявления динамически изменяющейся базы фактов используется специальный раздел – *FACTS* или *DATABASE*.

Раздел *GOAL* задает описание внутренней цели программы – предикат, элемент домена истинности, которого будет искать *Пролог.* Его называют *целевым предикатом*, или *целью*. Пролог осуществляет поиск только первого решения, для получения всех решений нужно предпринимать дополнительные действия.

## 3.6. Переменные в Прологе

В программах на языке Пролог существует три вида переменных:

1) *свободные* – пока не связана ни с каким значением;

2) *связанные* – переменная на время сопоставлена с элементами из домена истинности предиката;

3) *анонимные* – с этими переменными Пролог не связывает никаких значений, обозначаются "_".

При выполнении программы Пролога автоматически связывает и освобождает переменные. Имена переменных в Visual Prolog должны начинаться с **заглавной буквы** (**или с символа подчеркивания**). Переменные можно использовать в запросах.

**Пример 3.9.** Найти всех, кто любит теннис:

```
likes(Person, tennis).
```

В этом запросе переменная **Person** будет связываться с именами тех людей, которые согласно базе фактов любят **tennis**.

Для игнорирования ненужных доменов можно использовать анонимные переменные. Анонимная переменная может быть использована на месте любой другой переменной и ей никогда не присваивается значение. Анонимные переменные сопоставляются с любыми данными. Анонимные переменные также можно использовать в фактах.

**Пример 3.10. Ф**акты с использованием анонимных переменных:

У каждого есть туфли: `ows (_,shoes).`

Каждый ест: `eats(_).`

# 4. МЕХАНИЗМ ПОИСКА С ВОЗВРАТОМ

## 4.1. Сопоставление. Алгоритм унификации

В Пролог имеется несколько способов сопоставления объектов:

1. Идентичные структуры сопоставимы между собой, т. е. например, *родитель (иван, аня)* сопоставимо с *родитель (иван, аня)*.

2. Сопоставление с использованием свободных переменных. Например, если *X* свободна, то *родитель (иван, X)* сопоставимо с *родитель (иван, аня)* и *X* принимает значение «*аня»*. Если же *X* уже связана, то она действует так же, как обычная константа. Таким образом, если *X* связана со значением «*аня»*, то *родитель (иван, X)* сопоставимо с *родитель (иван, аня)*, но не сопоставимо с *родитель (иван, ира)*.

Две свободные переменные могут сопоставляться между собой. Такое связывание называют *совмещенными свободными переменными*. С момента связывания переменные трактуются как одна.

Поиск элемента домена истинности предиката и связывание компонент найденного элемента с аргументами предиката называют *согласованием предиката с базой знаний*. Алгоритм, который выпол-

няет согласование, – *алгоритм унификации*. Описание алгоритма унификации:

1. Конъюнкция (&) предикатов согласуется с БЗ слева направо, только успешно согласовав очередной предикат в конъюнкции, Пролог переходит к согласованию следующего предиката.

2. Если Пролог не смог найти элемент домена истинности очередного предиката в конъюнкции, что называется «потерпеть неудачу при согласовании», согласование оставшихся предикатов в конъюнкции прекращается.

3. При согласовании предиката Пролог просматривает факты и правила построчно сверху вниз, а в строке – слева направо.

4. Обратившись к очередному факту (правилу), Пролог помечает следующий одноименный факт (правило) так называемой **точкой отката**, создает копии переменных предиката и выполняет процесс унификации.

5. Потерпев неудачу, Пролог сначала **освобождает те переменные, которые он связал последними и потерпел неудачу**, затем обращается к точке отката, связывает освободившиеся переменные с новыми значениями и продолжает согласование.

6. Алгоритм унификации формирует наиболее общий унификатор для конечного множества унифицированных выражений.

Рассмотрим более детально схему работы интерпретатора Пролог и суть процесса унификации выражений на примере.

**Пример 4.1**

```
domains
   имя=symbol
predicates
   отец(имя,имя)
   дед(имя,имя)
clauses
   /*1*/ отец(алексей, владимир).
   /*2*/ отец(андрей, алексей).
   /*3*/ дед(X,Y):-отец(X,Z), отец(Z,Y).
goal
   /*4*/дед(X,Y).
```

В первом цикле интерпретатор просматривает фразы программы в последовательности 1–3, пытаясь унифицировать первый литерал из фразы 4 с первым литералом фраз 1-3. В БД только правило 3 может быть унифицировано с фразой 4. Перед унификацией интерпретатор переименовывает переменные правила 3 (*X,Y,Z*) в *X1, Y1, Z1* соответственно. Процесс унификации предполагает замену: *X* вместо *X1*, *Y*

вместо *Y1*. После сравнения литерала из стека с правилом 3 содержимое стека будет:

```
/*4а*/ отец (X,Z1), отец (Z1,Y).
```

Во втором цикле первый литерал (4а) унифицируется с фактом 1, «*алексей*» заменяет *X*, а «*владимир*» – *Z1*. После этого стек содержит:

```
/*4б*/ отец(владимир,Y).
```

В третьем цикле литерал (4б) не унифицируется ни с одним из первых литералов фраз 1, 2 или 3. Интерпретатор оказывается в тупике и поэтому выполняет возврат, т. е. содержимое стека, существовавшее до выполнения непосредственно предыдущего цикла, восстанавливается и принимает опять вид состояния 4а.

В четвертом цикле первый литерал (4а) унифицируется с фактом 2, «*андрей*» заменяется *X*, а «*алексей*» – *Z1*. Стек содержит:

```
/*4с*/ отец (алексей,Y).
```

В пятом цикле литерал (4с), оставшийся в стеке, унифицируется с фактом 1 и «*владимир*» заменяется *Y*. Стек вопросов становится пустым, что вынуждает интерпретатор выводить значения переменных, указанных в вопросе. В данном случае: *X=андрей, Y=владимир*.

Затем интерпретатор путем возврата пытается другими способами исчерпать стек вопросов. Но, начиная со стека содержимым в состоянии 4с, других возможных решений нет. Если не имеется других решений, то интерпретатор заканчивает работу.

## 4.2. Управление поиском решений

Рассмотрим процесс согласования целевого предиката. Для выполнения его подцелей Пролог начинает поиск с первого предложения, задающего предикат. Далее либо он находит соответствующее предложение, либо происходит следующее:

1. Если существует другое предложение, которое возможно может тоже удовлетворить подцель, Пролог размещает указатель **(точку отката**) рядом с соответствующим предложением.

2. Все свободные переменные в подцели связываются с соответствующими значениями в предложении.

3. Если соответствующее предложение является головой правила, то тело правила также должно быть удовлетворено.

Если соответствующие предложения невозможно найти, цель не выполняется. Пролог делает откат, поскольку пытается найти другое решение для предыдущей подцели. Когда будет достигнута последняя точка отката, Пролог освобождает все переменные и пытается заново выполнить первоначальный запрос.

Пролог начинает поиск с вершины программы. При совершении отката новый поиск начинается с последней точки отката. Если поиск неудачен, то опять происходит откат. Если все предложения будут перепробованы и не найдется решения для всех подцелей, основная цель считается невыполненной. Таким образом, можно выделить три основных правила поиска с возвратом:

1. Предикатные предложения проверяются в том порядке, в каком они появляются в программе, – сверху вниз.

2. Когда цель соответствует заголовку правила, выполняется согласование тела правила, которое образует новое множество подцелей для согласования.

3. Целевое утверждение считается согласованным, когда найдены факты для каждой подцели (листа) целевого дерева.

Встроенный механизм поиска с возвратом может привести к поиску ненужных решений либо, напротив, необходимо продолжать поиск дополнительных решений, даже если целевое утверждение уже согласовано. Пролог обеспечивает два инструментальных средства, управления механизмом поиска с возвратом:

1) предикат **fail**, для инициализации поиска с возвратом;

2) предикат **отсечение** (**!**) – для запрета возможности возврата.

Встроенный предикат **fai всегда** вызывает неуспешное завершение, и инициирует возврат, что позволяет найти все решения.

**Пример 4.2.** Пример применения предиката **fail**.

```
domains
   name = string
   sex = m ; f
predicates
   person(name,sex)
   writer
clauses
   person("Helen",f). person("Maggie",f).
   person("Suzanne",f). person("Petr",m).
   person("Alex",m).person("Mario",m).
   writer:- person(X,f),write(X),nl, fail.
Goal
writer(),nl.
```

Результат работы:

**с fail:**

```
Helen
Maggie
Suzanne
```

**без fail:**

```
Helen
```

Пролог предусматривает возможность прерывания поиска с возвратом с помощью отсечения (**!**). Пройдя через отсечение, уже невозможно произвести откат к подцелям, расположенным в обрабатываемом предложении перед отсечением, и также невозможно возвратиться к другим предложениям, определяющим обрабатывающий предикат (предикат, содержащий отсечение).

## 4.3. Повторы и рекурсия

Использование механизма поиска с возвратом позволяет определить все возможные решения целевого утверждения, также его можно использовать для выполнения итераций. Для этого необходимо задать предикат вида, реализующий бесконечное число откатов:

```
repeat.
repeat :- repeat.
```

**Пример 4.3.** Используя поиск с возвратом, реализовать поиск всех решений.

```
domains
   name = string
   sex = symbol
predicates
   person(name,sex)
   writer
   repeat
clauses
   person("Helen",f). person("Maggie",f).
   person("Suzanne",f). person("Petr",m).
   person("Alex",m). person("Mario",m).
   repeat.
   repeat :- repeat.
writer:- repeat,
write("Пол?"), readln(S), person(X,f), write(X),nl,
write("1-Следущее решение, 0-Выход:"), nl,
readchar(C),C = '0',!.
Goal
writer.
```

Результат работы:

```
Пол?f
1-Следущее решение, 0-Выход: Helen
1-Следущее решение, 0-Выход: Maggie
1-Следущее решение, 0-Выход: Suzanne
Пол?m
1-Следущее решение, 0-Выход: Petr
0
```

Рассмотрим работу программы более подробно:

1. ***repeat*** ставит точку отката.

2. Вводим пол человека, для поиска.

3. Находится первое согласование для ***person***, и ставится точка возврата, поскольку есть другие альтернативы. Найденное решение выводится на экран.

4. Вводим значение переменной ***C***, и если оно равно «***0***», то отсечение (снимаем все точки возврата) и завершение. В противном случае возвращаемся на последнюю точку возврата и находим следующее согласование для *person*. Когда все решения для выбранного пола найдены, поиск будет продолжен снова благодаря возврату оставленному ***repeat***.

Отличительная особенность такого способа заключается в том, что **все переменные теряют свои значения, когда обработка возвращается в точку отката, предшествующую тем вызовам предикатов, которые эти значения устанавливали**.

Еще одним способом реализации повторений являются рекурсивные правила. В такой процедуре нет проблемы запоминания результатов ее выполнения, поскольку вычисленные значения передаются из одного вызова в другой как аргументы рекурсивно вызываемого предиката.

Однако у рекурсии есть недостаток – она съедает память. При рекурсивном вызове информация о выполнении вызывающей процедуры должна быть сохранена для того, чтобы вызывающая процедура могла после выполнения вызванной процедуры возобновить выполнение. Для преодоления данного недостатка применяется ***хвостовая рекурсия***. В этом случае рекурсивный вызов является последней командой в правиле, что исключает необходимость вызывающей процедуре сохранять свое состояние, потому что эта информация уже не понадобится. Правила построения хвостовой рекурсии:

1. Рекурсивный вызов должен быть последней целью в хвостовой части правила вывода.

2. Перед рекурсивным вызовом не должно быть точек возврата.

Рассмотрим примеры ошибочных способов организации хвостовой рекурсии и способы ее устранения:

1. ***Непроверенная альтернатива к моменту выполнения рекурсивного вызова***. Стек должен быть сохранен, т. к. в случае неудачного завершения рекурсивного вызова вызывающая процедура может возвратиться и начать проверять эту альтернативу:

```
badcount1(X):-
     write(X),nl, NewX = X+1,
     badcount1(NewX).

badcount1(X):-
     X < 0, write("X отрицательное.").
```

Здесь первое предложение *badcount1* вызывает себя, когда вторая альтернатива еще не проверена. Такой вызов истощает память. **Отсечение** позволяет отвергать все возможные излишние альтернативы. Если в *badcount*1 переместить проверку из второго предложения в первое и добавить отсечение, то это позволит убрать точки возврата и сделать рекурсию оптимизированной:

```
cutcount1(X):-
     X>=0, !, write('\r',X),     NewX = X + 1,
     cutcount1(NewX).

cutcount1(_):-write("X отрицательное.").
```

2. ***Непроверенная альтернатива в любом вызываемом предложении предикате***.

```
badcount2(X):-
     write('\r',X),
     NewX = X+1,
     check(NewX),
     badcount2(NewX).
check(Z):-    Z >= 0.
check(Z):-    Z < 0.
```

Предложение *check* оставляет возможность альтернативны. Для решения проблемы необходимо сделать его детерминированным:

```
check(Z):- Z>0, !, write ("Положительное ").
check(_):- write ("Отрицательное ").
```

Рассмотрим примеры использования рекурсии.

**Пример 4.4.** Вычисление факториала.

**Вариант 1**

```
predicates
  factorial(integer, long)
  calc(integer, long, integer, long)
clauses
  factorial(N, FactN):- calc (N,FactN,1,1).
  calc (N, FactN, I, P) :-
     I <= N,!, NewP = P * I, Newl =I+1,
     calc (N, FactN, Newl, NewP).
calc (N, FactN, I, P):- FactN = P.

goal
factorial(10,X),write(X). %результат 3628800.
```

Предикат *factorial* задает начало рекурсии, а *calc* – рекурсивное правило. Первый вызов *factorial* унифицирует *N* и *FactN* и инициализирует значения для *I* и *P*. Пролог унифицирует переменную *FactN* в предложении *factorial*, с переменной, *FactN* правила *calc*, т. е. реально существует лишь одна *FactN*. Рассмотрим работу *calc*. Этот предикат проверяет предложение (условие циклического вычисления), а затем рекурсивно вызывает себя с новыми значениями для *I* и *P*. Здесь проявляется еще одна особенность Пролог. **Невозможно изменить значение переменной в Пролог присваиванием**. Вместо этого необходимо создать новую переменную и придать ей нужное значение: `NewP = P + 1.`

В этом предложении отсечение будет обеспечивать оптимизацию хвостовой рекурсии. В конечном счете *I* превысит *N*, текущие значения *P* и *FactN* унифицируются благодаря второму правилу *calc* и рекурсия прекратится. Более красиво второе правило можно записать как: `calc(_, FactN, _, FactN).`

**Вариант 2**

```
predicates
    factorial(integer, long)
    factorial(integer, long, integer, long)
clauses
    factorial(N,FactN):-factorial(N,FactN,1,1).
    factorial(N,FactN,N,FactN):-!.              %1
    factorial(N,FactN,I,P):-                        %2
    NewI = I+1,
    NewP = P*NewI,
    factorial(N, FactN, NewI, NewP).      %3
goal
factorial(10,X), write(X). % результат 3628800.
```

Порядок предикатов *factorial* важен. Передача результата происходит при согласовании предиката *%1*. Сначала Пролог пытается сопоставить *%1* – сравниваются *FactN* с *NewP* , но поскольку *NewI* <> *N* то снова вызывается *%2*. По достижении *NewI* = *N* предикат *%1* успешно согласуется, а отсечение обеспечивает выход из рекурсии.

**Пример 4.5.** Преобразование данных в зависимости от выбранной операции.

```
predicates
   menu(integer)
   action(char, integer, integer)
clauses
   menu(Old):-nl,
   write("1-Сложение\n"),
```

```
   write("2-Вычитание\n"),
   write("0-Выход\n"),
   write("Enter number - "),
   readchar(X),nl,
   action(X,Old,New),!,
   menu (New).
   action('1',Old,New):-                          %1
   write("Слагаемое:"),readint(Val),
   New=Old+Val, write("Value=",New),!.
   action('2',Old,New):-                          %2
   write("Вычитаемое:"),readint(Val),
   New=Old-Val, write("Value=",New),!.
   action('0', _, _):-exit.                       %3
   action(_, New, New).                           %4
goal
   menu(0).
```

Правило *menu* – реализует меню с хранением текущего значения модифицируемого параметра. Предложения *%1*, *%2 action* – модифицирует параметр *Old* в зависимости от выбранной операции и возвращают результат через переменную *New*. Предложение *%4* выполняет копирование параметра, если выбран не существующий пункт меню.

# 5. СПИСКИ, ДЕРЕВЬЯ, ГРАФЫ

## 5.1. Списки

### *5.1.1. Задание списков*

В Пролог списком называется конечная последовательность элементов одного домена. Элементы списка записываются в квадратных скобках через запятую. Список, не содержащий элементов, называется *пустым* и обозначается [5]. Общий вид объявления:

```
имя_списка = имя_домена_элементов *
```

**Пример 5.1.** Пример объявления списка целых чисел:

```
domains
   intlist = integer*
```

Для объявления списка, состоящего из элементов разных доменов (объединения), необходимо определить домен, включающий необходимые типы с функторами, указывающими тип элемента.

**Пример 5.2.** Пример объявления списка, элементы которого могут быть целыми или вещественными числами, а также строками:

```
domains
list = l(list); i(integer); c(char); s(string)
my_list = list*
[2, 9, ["food", "goo"], "new"]  %Неверное описание
```

```
[i(2), i(9), l([s("food"), s("goo")]), s("new")] % Кор-
ректно
```

Список является рекурсивным составным объектом. Он состоит из двух частей – головы (первый элемент) и хвоста, который является списком, включающим все последующие элементы (табл. 5.1). *Хвост списка – всегда список, голова списка – всегда элемент.* Для отделения головы списка от хвоста используется символ вертикальная черта «|».

*Таблица 5.1*

**Пример 5.3.** Головы и хвосты списков

| **Список** | **Голова** | **Хвост** |
|---|---|---|
| ['a', 'b', 'c'] | 'a' | ['b', 'c'] |
| ['a'] | 'a' | [] – пустой список |
| [] | Не определена | Не определен |
| [[1, 2, 3],[2, 3, 4], []] | [1, 2, 3] | [[2, 3, 4], []] |

`[a, b, c]` эквивалентно `[a | [b, c]]`;
`[a | [b, c]]` эквивалентно `[a| [b | [c]]]`;
`[a| [b | [c]]]` эквивалентно `[a | [b | [c | []]]]`.

Если рекурсивно выбирать первый элемент списка, то последним элементом всегда будет пустой список []. Пустой список нельзя разделить на голову и хвост. Как и другие составные объекты, список имеет структуру дерева.

### *5.1.2. Основные алгоритмы обработки списков*

Список является рекурсивной составной структурой данных. Классический способ обработки – выделение головы списка, пока хвост не будет пустым списком. Обычно алгоритм состоит из двух предложений: первое из них говорит, что делать со списком, который можно разделить на голову и хвост, второе – что делать с пустым списком.

**Пример 5.4.** Ввод–вывод элементов списка. Алгоритм формирования списка из вводимых с клавиатуры символов:

1. Непустой символ ставим в голову списка – результата и продолжаем формирование списка.

2. Условие выхода из рекурсии – $H = 0$, второе предложение для успешного согласования предиката в целом.

Алгоритм вывода списка:

1. Если список пустой, то ничего не делать.

2. Иначе, разбить список на голову и хвост, голову – вывести на экран, а затем печатать хвост (список).

```
domains
list = integer*
predicates
write_a_list(list)
get_list(list)
clauses
get_list([H|T]) :-
        readint(H), H <> 0, !, get_list(T).
   get_list([]).
   write_a_list([]).
   write_a_list([H|T]):-
        write(H), nl, write_a_list(T).
goal
get_list(List), write_a_list(List).
```

**Пример 5.5.** Подсчет числа элементов списка (хвостовая рекурсия). Алгоритм определения числа элементов списка:

1. Длина пустого списка [ ] – 0.

2. Длина любого другого списка – 1 плюс длина его хвоста.

Описание аргументов предиката *length*:

– исходный список;

– длина списка;

– счетчик, увеличивающийся на 1 при каждом вызове.

```
domains
  list = integer*
predicates
  length(list, integer, integer)
clauses
  length([], Result, Result).
  length([_|T], Result, Counter) :-
  NewCounter = Counter + 1,
  length (T, Result, NewCounter).
goal
  length([1, 2, 3], L, 0),    %начать со счетчика = 0
  write("L=",L), nl.
```

**Пример 5.6.** Преобразование элементов списка. Алгоритм формирования:

1. Если список пустой, то результат – пустой список.

2. Если не пустой, то разделить исходный список на голову (*Head*) и хвост (*Tail*). Модифицировать (*Head*) и результат сделать головой нового списка (*Head1*).

3. Рекурсивно изменить каждый элемент хвоста списка (*Tail*) и сделать его хвостом результата (*Tail1*).

```
domains
list = integer*
predicates
   add(list, list)
clauses
   add([],[]). % граничное условие
   add([Head|Tail],[Head1|Tail1]):- %отделить голову
   Head1= Head+1,   % добавить 1 к голове
   add(Tail,Tail1).      % обработать хвост
goal
add ([1,2,3,4], NewList).
```

**Пример 5.7.** Формирование нового списка из положительных элементов исходного. Алгоритм:

1. Если список пустой, то результат – пустой список.

2. Если голова списка *H* отрицательная, то продолжить разбор хвоста *T*.

3. Если предыдущее правило ложно, ставим голову списка *H* в начало формируемого и продолжаем разбор хвоста *T*.

```
new_list([], []).
new_list ([H|T], Tail):-
   H < 0, !,  % если H отрицательно, то пропустить
   new_list (T, Tail).
new_list([H|T], [H|Tail]):- %H в голову нового списка
   new_list (T, Tail).
```

**Пример 5.8.** Проверка на принадлежность списку. Алгоритм:

1. Элемент принадлежит списку, если он равен элементу головы.

2. Иначе продолжить поиск в хвосте списка.

```
domains
namelist = name*
name = symbol

predicates
member(name, namelist)

clauses
member(Name,[Name|_]):-!.
member(Name,[_|Tail]):- member(Name, Tail).
```

Предикат можно использовать по-разному:

`member(2, [1, 2, 3, 4]).` – проверить 2 входит в список

`member(X, [1, 2, 3, 4]).` – найти всех членов списка

**Пример 5.9.** Объединение списков. Алгоритм объединения:

1. Слияние пустого списка с любым списком *L* дает список *L*.

2. Если первый список не пустой, то голову первого списка ставим в голову списка – результата и продолжаем слияние хвоста первого списка со вторым списком.

```
domains
integerlist  = integer*
predicates
append(integerlist, integerlist, integerlist)
clauses
   append(List, [], List):-!.
   append([], List, List):-!.
   append([H|L1], List2, [H|L3]):-
      append(L1, List2, L3).
```

Способы использования:

1. `append ([1, 2, 3], [5, 6], L).`

2. `append ([1, 2], [3], L), append(L, L, LL).`

3. `append(L1, L2, [1, 2, 4]).` – из каких списков можно получить такой результат.

Решения:

```
L1 =[], L2=[1, 2, 4]
L1=[1], L2=[2, 4]
L1=[1, 2], L2=[4]
L1=[1, 2, 4], L2=[]
```

4. `append (L1, [3, 4], [1, 2, 3, 4]).` – какой список нужно подсоединить к [3, 4] для получения списка [1, 2, 3, 4].

`L1=[1, 2].`

**Пример 5.10.** Удаление заданного элемента из списка. Алгоритм (удаление первого найденного):

1. Если удаляемый элемент *H* является головой списка, то результатом удаления будет хвост списка.

2. Если удаляемый элемент *H* не является головой списка, то голову исходного списка ставим в голову списка – результата и удаляем *H* из хвоста списка.

```
delete(H, [H | Tail], Tail).
delete(H, [Y | Tail], [Y|Tail_1]):-
delete(H, Tail, Tail_1).
```

Алгоритм удаления всех вхождений заданного элемента в список:

```
delete(_, [], []).
delete(H, [H | Tail], Tail_1):-
!, delete(H, Tail, Tail_1).
```

```
delete(H, [Y | Tail], [Y|Tail_1]):-
!, delete(H, Tail, Tail_1).
```

### *5.1.3. Формирование списка результатов*

Рекурсия не обеспечивает поиск альтернативных решений в целевом утверждении. Для этого можно использовать встроенный предикат ***findall***, который собирает все решения для целевого утверждения в единый список. Синтаксис предиката ***findall***:

```
findall(<VAR_DOMAIN>VarName, myPredicate (аргументы),
<VAR_DOMAIN_LIST>ListParam),
```

где

1) *VarName* – параметр, который необходимо собрать в список;

2) *myPredicate* – целевое утверждение, решение которого собираем в список;

3) *ListParam* – список-результат.

При использовании предиката *findall* необходимо объявить в программе домен элементов списка результата.

```
domains
var_domain_list = var_domain*
```

**Пример 5.11.** Найти все варианты списков, из которых можно получить исходный.

```
domains
    integerlist = integer*
    llist = integerlist*
predicates
    append(integerlist, integerlist, integerlist)
clauses
    append([], List, List).
    append([H|L1], List2, [H|L3]):-
    append(L1, List2, L3).
goal
    findall(Z, append(Z, Y, [1, 2, 4]), L1) ,
    findall(Y, append(Z, Y, [1, 2, 4]), L2) ,
    nl,write(L1,L2).
```

Результат:

```
L1=[[],[1],[1,2],[1,2,4]].
L2=[[1,2,4],[2,4],[4],[]].
```

## 5.2. Деревья

### *5.2.1. Рекурсивные структуры данных*

***Графом*** называют пару множеств вершин и дуг. Различают ориентированные и неориентированные графы. В ориентированном графе каждая дуга имеет направление (рассматриваются упорядоченные пары вершин). ***Дерево*** – граф, у которого одна вершина корневая, остальные вершины имеют только одного отца и все вершины являются потомками корневой вершины. ***Двоичным*** называется дерево, все значения которого в левом поддереве меньше корня, а значения в вершинах правого больше корневого значения. Левое и правое поддеревья также являются двоичными деревьями. Такие деревья еще называют *упорядоченными слева направо*. ***Путь*** – последовательность вершин, соединенных дугами. Для ориентированного графа направление пути должно совпадать с направлением каждой дуги, принадлежащей пути. ***Листом*** дерева называется его вершина, не имеющая сыновей. ***Кроной*** дерева называется совокупность всех листьев. ***Высотой*** дерева называется наибольшая длина пути от корня к листу.

Пролог позволяет задавать рекурсивные структуры, причем указатели создаются и обслуживаются автоматически. Дерево можно задать, определив домен вида:

```
DOMAINS
treetype = tree(char, treetype, treetype) ; empty
```

Домен *treetype* определен как функтор – ***дерево*** **(*tree*)**, аргументы которого – вершина и две ветви (поддерева). Функтор *empty* предназначен для обеспечения способа прекращения рекурсии. Он определяет оконечные листы дерева. Рассмотрим основные алгоритмы работы с деревьями.

### *5.2.2. Двоичные деревья*

Для определения домена описания дерева воспользуемся рекурсивным определением: дерево либо пусто, либо состоит из корня, левого и правого поддеревьев, которые, в свою очередь, также являются деревьями. Для ввода двоичного дерева воспользуемся предикатом ***insert*****,** выставляющим элемент в **левое** поддерево, если он **меньше** корневого, или в **правое** в противном случае.

**Пример 5.12.** Построение двоичного дерева с вершинами числового типа. Значения вершин генерируются случайным образом.

```
domains
    tree = tree(integer, tree, tree); end
```

```
predicates
     insert(integer, tree, tree)
     write_tree(tree)
     tree_gen(integer, tree)
clauses
     insert(New,end,tree(New,end,end)):-!.
     in-
sert(New,tree(Num,Left,Right),tree(Num,NewLeft,Right)):-
     New<Num, !, insert(New,Left,NewLeft).
     in-
sert(New,tree(Num,Left,Right),tree(Num,Left,NewRight)):-
insert(New,Right,NewRight).

     write_Tree(end).
     write_Tree(tree(Item,Left,Right)):-
     write_Tree(Left),       write(Item,       "       "),
write_Tree(Right).

tree_gen(0,end):-!.
tree_gen (N,T):-
     random(100,X), N1= N-1, tree_gen (N1,T1),
     insert(X,T1,T).
goal
tree_gen(10, Tree), write_Tree(Tree).
```

Внутреннее представление дерева в Пролог:

```
tree(25,tree(22,end,end),tree(76,end,tree(91,end,end)))
```

### *5.2.3. Принадлежность дереву*

Зададим предикат, осуществляющий поиск значения в дереве. Предикат реализуется следующим алгоритмом:

1. **Базис рекурсии**: значение принадлежит дереву, если оно содержится в корне дерева.

2. **Шаг рекурсии**: искать в левом поддереве, либо в правом.

**Пример 5.13.** Предикат имеет два аргумента: первый значение для поиска, второй - дерево поиска.

```
     tree_member(X,tree(X,_,_)):-!.
tree_member(X,tree(_,L,_)):- tree_member(X,L),!.
     tree_member(X,tree(_,_,R)):- tree_member(X,R).
```

Усовершенствуем предикат для проверки принадлежности значения двоичному дереву. Повысить эффективность можно за счет отсечения. Если искомое значение меньше корневого, то его следует искать только в левом поддереве, если больше, то только в правом поддереве.

**Пример 5.14.** Поиск в двоичном дереве

```
tree_member2(X, tree(X,_,_)):-!. /* X - корень дерева */
tree_member2(X, tree(K,L,_)):-   /* X- в левом поддереве */
X<K,!, tree_member2(X,L).
tree_member2(X, tree(K,_,R)):-   /* X- в правом поддереву
*/
tree_member2(X,R).
```

### *5.2.4. Замена элемента в дереве*

Замена в дереве заданного элемента другим реализуется следующим алгоритмом:

1. **Базис рекурсии**: в пустом дереве нет элементов.

2. **Шаг рекурсии** зависит от того, где находится заменяемое значение: если в корне дерева, то его нужно заменить на заданное и перейти к замене в левом и правом поддереве. Если значение в корне отличное от заменяемого, то оно остается, а замена производится в поддеревьях.

Данный алгоритм не сохраняет двоичную структуру дерева.

**Пример 5.15.** Предикат замены элемента в дереве. Аргументы: три входных (значение, которое нужно заменять; значение, которым нужно заменять; исходное *дерево*), четвертый – выходной – полученное *дерево*.

```
tree_replace(_,_,empty,empty). /* пустое остается пустым */
     /* корень содержит заменяемое значение X*/
tree_replace(X,Y,tree(X,L,R),tree(Y,L1,R1)):-
     /* L1 - результат замены в дереве L всех вхождений X на
Y */
     !,tree_replace(X,Y,L,L1),
     /* R1 - результат замены в дереве R всех вхождений X на
Y */
     tree_replace(X,Y,R,R1).

     /* корень не содержит заменяемое значение X */
     tree_replace(X,Y,tree(K,L,R),tree(K,L1,R1)):-
     /* L1 - результат замены в дереве L всех вхождений X на
Y */
     tree_replace(X,Y,L,L1),
     /* R1 - результат замены в дереве R всех вхождений X на
Y */
     tree_replace(X,Y,R,R1).
```

### *5.2.5. Удаление элемента в двоичном дереве*

При удалении элемента дерева, не являющегося листом (внутренняя вершина), возникает проблема с заполнением пустого места после удаления элемента (рис. 5.1).

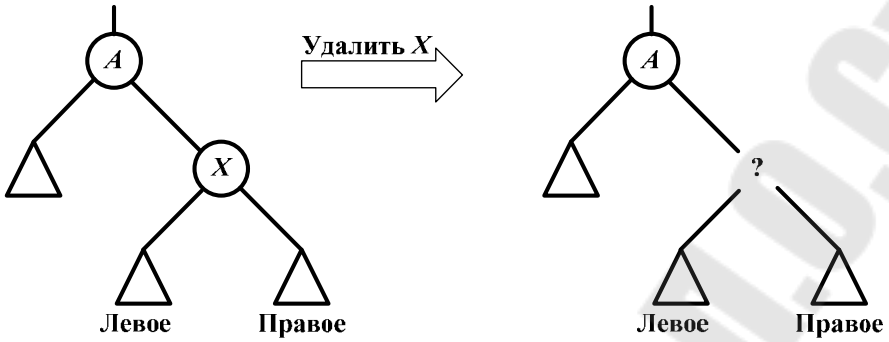



*Рис. 5.1*.Удаление *X* из двоичного дерева

Для разрешения проблемы можно удалить из правого поддерева минимальный элемент (или из левого дерева максимальный) и заменить им значение, находящееся в корне. Так как любой элемент правого поддерева больше любого элемента левого, то результирующее останется двоичным (рис. 5.2).

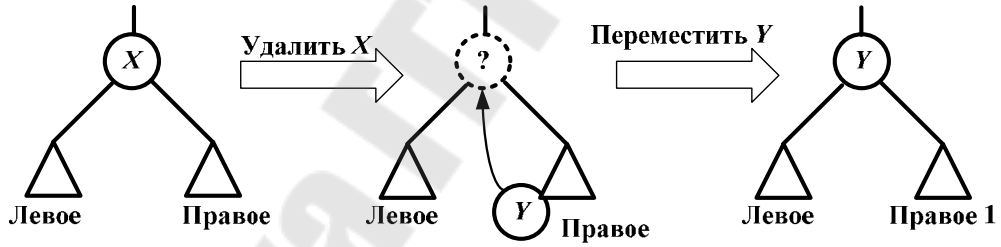



*Рис. 5.2*. Заполнение пустого места после удаления *X*

Для удаления вершины, содержащей минимальный элемент, зададим вспомогательный предикат *tree_del*. Алгоритм состоит из двух предложений.

1. **Базис рекурсии**: если левое поддерево пусто, минимальный элемент дерева – значение в корне (в правом элементы большие вершины), значит – удалить корень, а результат – правое поддерево.

2. **Шаг рекурсии**: в противном случае продолжаем поиск минимального элемента в левом поддереве.

**Пример 5.16.** Удаление элемента в двоичном дереве.

```
/* Если левое поддерево пусто, то минимальный элемент -
корень, а дерево без минимального элемента - правое подде-
рево.*/
```

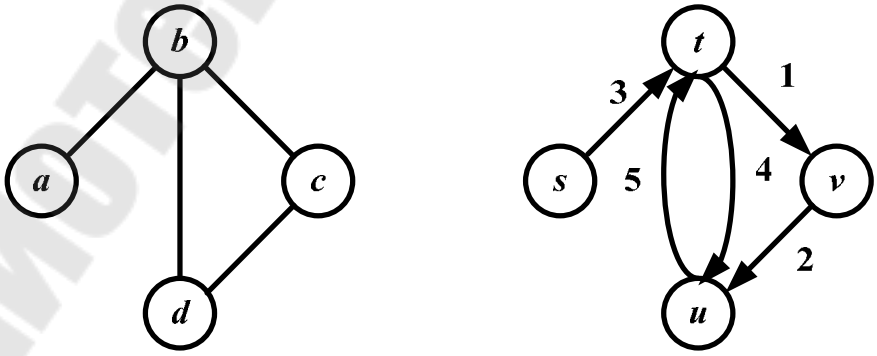
```
    tree_del_min(tree(X,end,R), R, X).
    /* Левое поддерево не пусто, значит, оно содержит ми-
нимальное значение всего дерева, которое нужно удалить */
tree_del_min(tree(K,L,R), tree(K,L1,R), X):-
    tree_del_min(L, L1, X).

    /* X совпадает с корневым значением исходного дерева,
и левое поддерево пусто */
tree_delete(X,tree(X,end,R), R):-!.
    /* X совпадает с корневым значением исходного дерева,
и правое поддерево пусто */
tree_delete (X,tree(X,L,end), L):-!.
    /* X совпадает с корневым значением исходного дерева,
причем ни левое, ни правое поддеревья не пусты */
tree_delete (X,tree(X,L,R), tree(Y,L,R1)):-
    tree_del_min(R,R1,Y).
    /* X меньше корневого значения дерева */
tree_delete (X,tree(K,L,R), tree(K,L,R1)):-
    /* X больше корневого значения дерева */
tree_delete (X,tree(K,L,R), tree(K,L1,R)):-
    X<K,!, tree_delete (X,L,L1).
    tree_delete (X,R,R1).
```

## 5.3. Графы

### *5.3.1. Представление графов*

Граф определяется как множество *вершин* и *ребер*, каждое ребро задается парой вершин. Направленные ребра называют *дугами*, а графы – *направленными*. Ребрам можно приписывать стоимости, имена или метки произвольного вида, в зависимости от конкретного приложения (рис. 5.3).




*а*) *б*)

*Рис. 5.3*. Графы:
*а* – обычный; *б* – направленный, со стоимостью дуг

В Прологе графы можно представлять различными способами. Графы рис. 5.3 можно представить в виде следующего множества предложений:

1. Множеством предложений:

```
связь(a, b). связь(b, c). связь(b, d). связь(c, d).
ду-
га(s,t,3).дуга(t,v,1).дуга(u,t,2).дуга(t,u,5).дуга(v,u,2).
```

2. Списочное описание дуг:

```
G1=граф([a, b, c, d],[d( a, b), d( b, d), d( b, c), d( c,
d)])
где дуга задана функтором – d=d(symbol, symbol)
G2=граф([s,t,u,v],[d(s,t,3),d(t,v,1),d(t,u,5),d(u,t,2),d(v
,u,2)]), где дуга задана функтором – d=d(symbol, symbol,
integer).
```

3. Списочное описание смежных вершин:

```
G1 = [[a,[b]], [b,[a, c, d]], [c,[b, d]], [d,[b, c]]].
```

### *5.3.2. Поиск пути в графе*

Зададим предикат, определяющий путь в графе *G* между двумя заданными вершинами *A* и *Z*. Определим отношение:

```
путь(A, Z, G, P),
```

где ***G*** – граф; ***A*** и ***Z*** – две его вершины; ***P*** – ациклический путь между ***A*** и ***Z*** в графе ***G***.

Так, для графа, изображенного на рис. 5.3, *а*, верно:

```
путь(a, d, G, [a, b, d])
путь(a, d, G, [a, b, c, d])
```

Поскольку путь не должен содержать циклов, любая вершина может присутствовать в пути не более одного раза. Зададим алгоритм поиска ациклического пути *P* между *A* и *Z* в графе *G*:

1. **Базис**: **Если *A* = *Z*** (конечная точка), то путь ***P* = *[A].***

2. **Рекурсивное правило**: иначе найти ациклический путь *P1* из произвольной вершины *Y* в *Z*, а затем найти путь из *A* в *Y*, не содержащий вершин из *P1* (рис. 5.4).

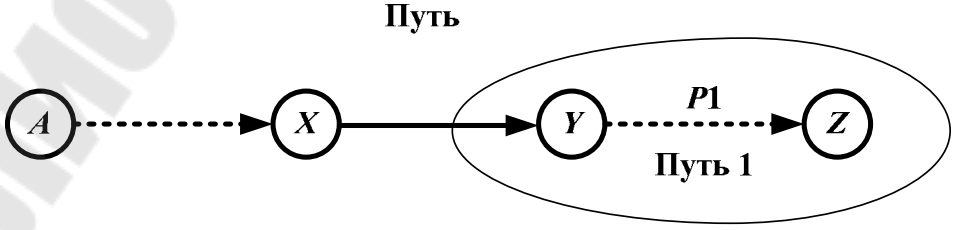



*Рис. 5.4*. Отношение **путь1**: Путь – это путь между *A* и *Z*, в своей заключительной части он перекрывается с **Путь1**

В таком алгоритме неявно предполагается дополнительное отношение: путь из *A* в *Y* не должен проходить через вершины из подмножества *P1*. В связи с этим мы определим еще одну процедуру:

```
путь1(A, P1, G, P),
```

где *A* – некоторая вершина; *G* – граф; *P1* – путь в *G*; *P* – ациклический путь в *G*, идущий из *A* в начальную вершину пути *P1*, а затем – вдоль пути *P1* вплоть до его конца.

Между **путь** и **путь1** имеется следующее базовое соотношение:

```
путь(A,Z,Граф,Путь):-путь1(A,[Z],Граф,Путь).
```

Зададим алгоритм отношения **путь1**.

1. **Базис рекурсии** (граничное условие): когда начальная вершина пути *P1 – Y* совпадает с начальной вершиной *A* пути *P*.

2. **Рекурсивное правило**: если же начальные вершины путей не совпадают, то найти вершину *X*, смежную с *Y* (решение не содержится в *P1*).

```
путь1(A,[A|Путь1],_,[A|Путь1]).
путь1(A,[Y|Путь1],Граф,Путь):-
смеж(X,Y,Граф),
not(принадлежит(X,Путь1)),
путь1(A,[X,Y|Путь1],Граф,Путь).
```

Таким образом, путь ***P1*** постепенно расширяется, пока не достигнет начала ***A***. Рассмотренные правила позволяют найти все возможные варианты путей. Вспомогательные отношения:

***смеж(X, Y, G)*** – определяет в графе ***G*** вершину, смежную с ***X*** и наоборот. Определение этого отношения зависит от способа представления графа. Будем использовать представление графа как пара множеств вершин и ребер. Тогда

```
смеж(X,Y,graf(Верш,Реб)):-
принадлежит(d(X,Y),Реб);
принадлежит(d(Y,X),Реб).
```

**Пример 5.17.** Поиск пути в графе между вершинами

```
domains
nodes_list=symbol*
d=d(symbol,symbol)
d_list=d*
graf=graf(nodes_list,d_list);empty
predicates
принадлежит(symbol,nodes_list)
принадлежит(d,d_list)
смеж(symbol,symbol,graf)
путь(symbol,symbol,graf,nodes_list)
путь1(symbol,nodes_list,graf,nodes_list)
```

```
clauses
     принадлежит(Name,[Name|_]).
     принадлежит(Name,[_|Tail]):-принадлежит(Name,Tail).
     смеж(X,Y,graf(Верш,Реб)):-
     принадлежит(d(X,Y), Реб) ; принадлежит(d(Y,X), Реб).
     путь(A,Z,Граф,Путь):- путь1(A,[Z],Граф,Путь).
путь1(A,[A|Путь1],_,[A|Путь1]).
путь1(A,[Y|Путь1],Граф,Путь):-
     смеж(X,Y,Граф), not(принадлежит(X,Путь1)),
     путь1(A,[X,Y|Путь1],Граф,Путь).
goal
G1=graf([s,t,u,v],[d(s,t),d(t,v),d(t,u),d(u,t),d(v,u)]),
путь(u,t,G1,Out), write(Out),nl,fail.
```

Результат поиска пути из *U* в *T* (рис. 5.3, *б*, считаем граф не направленным): `["u","t"], ["u","v","t"], ["u","t"].`

### *5.3.3. Поиск пути в графе со стоимостью*

Каждому пути в графе можно задать стоимость, или длину дуги. Модифицируем отношения путь и путь1. Введем дополнительный аргумент *C* – стоимость пути:

```
путь( A, Z, G, P, C) C - стоимость пути P
путь1( A, P1, C1, G, P, C) C1 - стоимость пути P1
```

**Пример 5.18.** Поиск пути в графе между вершинами с вычислением его стоимости.

```
     domains
     nodes_list=symbol*
     d=d(symbol,symbol, integer)
     d_list=d*
     graf=graf(nodes_list,d_list);empty
predicates
     принадлежит(symbol,nodes_list)
     принадлежит(d,d_list)
     смеж(symbol,symbol,graf,integer)
     путь(symbol,symbol,graf,nodes_list,integer)
     путь1(symbol,nodes_list,integer,graf,nodes_list,integer)
clauses
     принадлежит(Name,[Name|_]).
     принадлежит(Name,[_|Tail]):- принадлежит(Name,Tail).

     смеж(X,Y,graf(Верш,Реб),Стоим):-
     принадлежит(d(X,Y,Стоим),Реб) ;
     принадлежит(d(Y,X,Стоим),Реб).

     путь(A,Z,Граф,Путь,Ст) :-путь1(A,[Z],0,Граф,Путь,Ст).
     путь1(A,[A|Путь1],Ст1,_,[A|Путь1],Ст1).
```

```
    путь1(A,[Y|Путь1],Ст1,Граф,Путь,Ст) :-
    смеж(X,Y,Граф,СтXY),
    not(принадлежит(X,Путь1)), Ст2=Ст1+СтXY,
    путь1(A,[X,Y|Путь1],Ст2,Граф,Путь,Ст).
goal
    G1=graf([s,t,u,v],  [d(s,t,3),  d(t,v,1),  d(t,u,5),
d(u,t,2),  d(v,u,2)]),  путь(t,u,G1,Out,Ст),  write(Out),
write("=",Ст),fail.
```

Результат:

```
["t","u"]=5, ["t","v","u"]=3, ["t","u"]=2.
```

Для использования направленного графа необходимо модифицировать отношение ***смеж***, в котором следует убрать второе отношение, задающее перестановку.

```
    смеж(X,Y,graf(Верш,Реб),Стоим):-
принадлежит(d(X,Y,Стоим),Реб).
    goal
    G1=graf([s,t,u,v],[d(s,t,3),d(t,v,1),d(t,u,5),d(v,u,2
)]),
    путь(s,u,G1,Out,Ст),
write(Out),write("=",Ст),nl,fail.
```

Результат: `["s","t","u"]=8, ["s","t","v","u"]=6.`

# 6. СТРАТЕГИИ ПОИСКА РЕШЕНИЙ

## 6.1. Пространство состояний

Общей схемой формального представления задач является пространство состояний. ***Пространство состояний*** – это направленный граф, вершины которого соответствуют проблемным ситуациям, а дуги – возможным ходам. Конкретная задача в заданном пространстве определяется *стартовой вершиной* и *целевым условием*. ***Решение задачи*** сводится к поиску пути в графе. Оптимизационные задачи моделируются приписыванием каждой дуге пространства состояний некоторой стоимости.

Пространство состояний может задаваться явным образом в виде фактов или при помощи правил вычисления вершин-преемников некоторой заданной вершины. Второй способ является гораздо более практичным и позволяет описывать сложные и громоздкие схемы переходов. Будем представлять пространство состояний отношением **после (*X*, *Y*)** или **после (*X*, *Y*, *Ст*)** – которое истинно в том случае, если в пространстве состояний существует разрешенный ход из вершины ***X*** в вершину ***Y***. В этом случаи ***Y*** – это преемник вершины ***X***, ***Ст*** – стоимость перехода.

Другой задачей является представление множеством переходов. Обычно используется списочное, или древовидное описание. Второй способ является более компактным, но и более сложным в реализации.

Имеются две основных стратегии поиска в пространстве состояний – *поиск в глубину* и *поиск в ширину*.

## 6.2. Стратегия поиска в глубину

Стратегия поиска в глубину, решающего пути ***Реш*** из заданной вершины ***B*** в некоторую целевую вершину, реализуется следующим алгоритмом:

1. Базис рекурсии: если ***B*** – это целевая вершина, то положить ***Реш = [B]***.

2. Шаг рекурсии: если для исходной вершины ***B*** существует вершина – преемник ***B1***, такая, что можно провести путь ***Реш1*** из ***B1*** в целевую вершину, то положить ***Реш = [B | Реш1]***.

Рассмотрим реализацию поиска в глубину для пространства состояний, описанного в виде фактов (рис. 6.1).

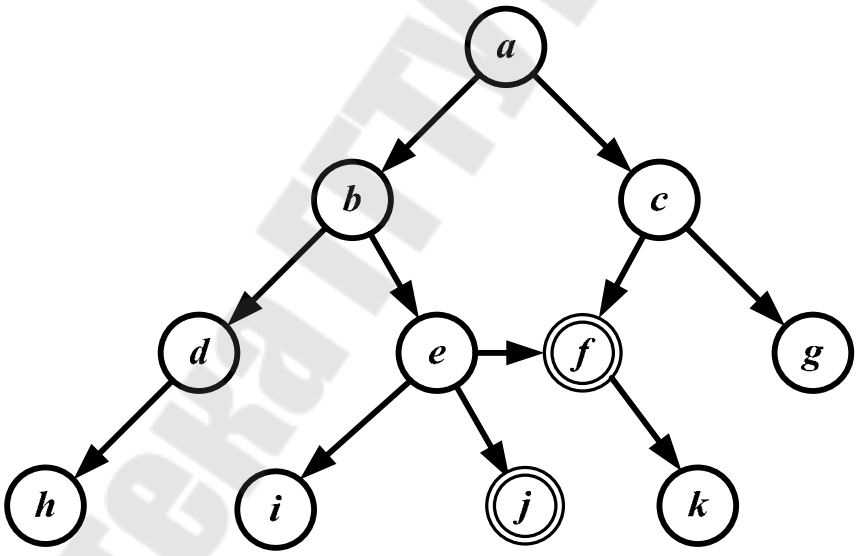



*Рис. 6.1*. Пример пространства состояний

**Пример 6.1.** Для пространства состояний, представленого рис. 6.1, реализовать поиск в глубину. Задание связей фактом *потомок – преемник*.

```
domains
list=symbol*
predicates
    решить1(symbol, list)
    после(symbol, symbol)
```

```
     цель(symbol)
clauses
     после(a,b). после(a,c). после(b,d). после(b,e).
     после(d,h). после(e,i). после(e,j). после(e,f).
     после(c,f).после(c,g). после(f,k).

     решить1(Верш, [Верш]):- цель(Верш).

решить1(Верш, [Верш|Реш1]):-
     после(Верш, Верш1), решить1(Верш1, Реш1).
```

При поиске в глубину для бинарного дерева из возможных альтернатив в пространстве состояний всегда выбирается самая «глубокая», т. е. вершина, расположенная дальше других от стартовой. Так, для цели:

```
     цель(j).
     goal
     решить1(a,T), write(T),nl, fail.
```

Порядок просмотра вершин будет следующим: *a*, *b*, *d*, *h*, *e*, *i*, *j* (рис. 6.2), а найденное решение: [*"a","b","e","j"*].

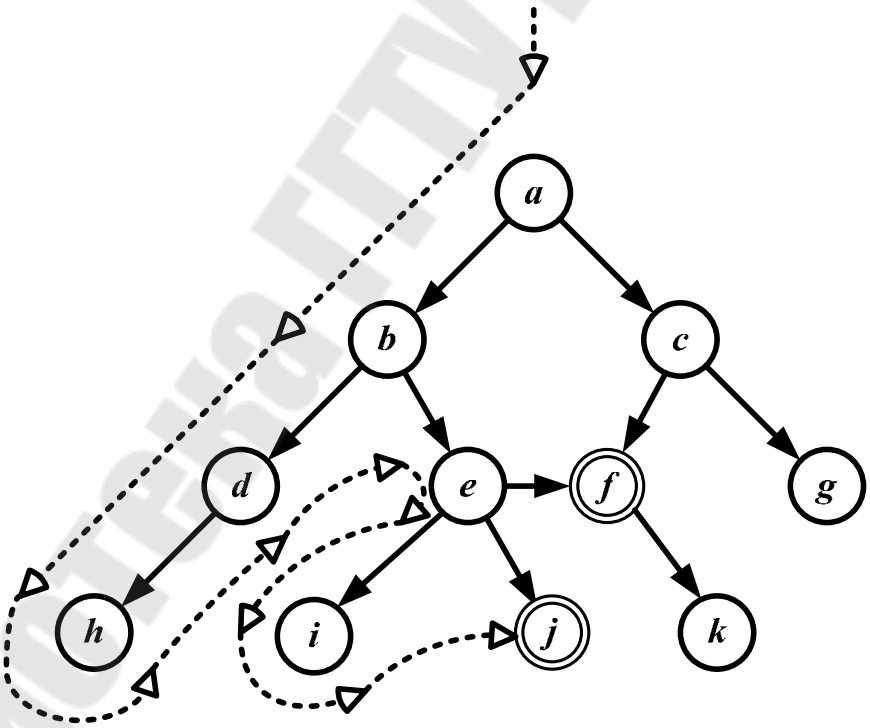



*Рис. 6.2*. Пример поиска в глубину:
*a* – стартовая вершина; *j* – целевая вершина

Если же задать целевой вершиной ***k***, то найденных решений будет несколько: *["a","b","e","f","k"] ["a","c","f","k"]*.

Рассмотренный алгоритм поиска в глубину не имеет защиты от циклических связей в пространстве состояний. Так, например, если добавить дугу, ведущую из *h* в *d* (рис. 6.3), то поиск «зациклится» поскольку произойдет *не возврат* из *h*, а снова *переход* к *d* и снова к своему приемнику *h*.

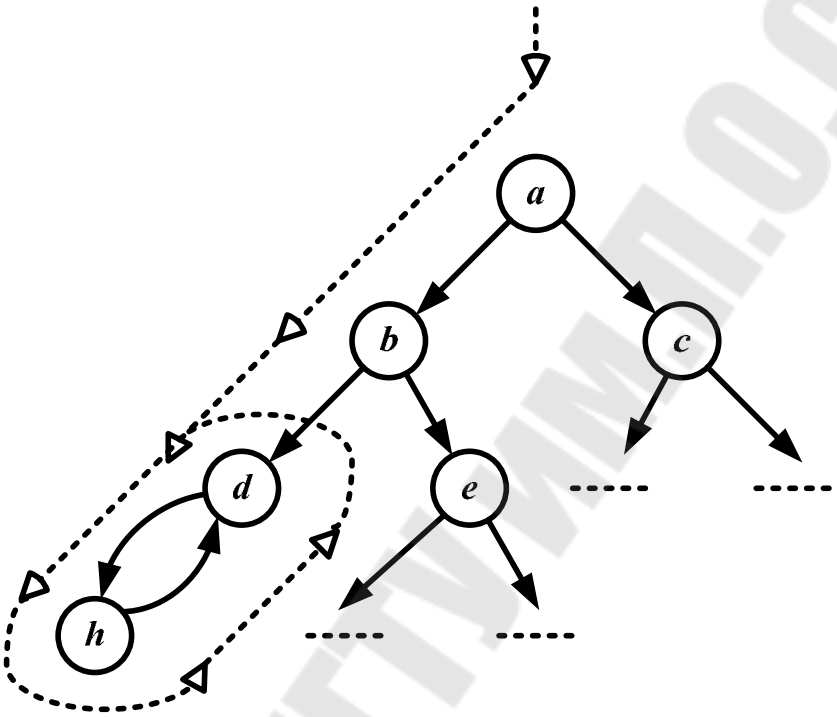



*Рис. 6.3*. Начинаясь в *a*, поиск в глубину заканчивается бесконечным циклом между *d* и *h*: *a*, *b*, *d*, *h*, *d*, *h*, *d*...

Усовершенствованием алгоритма поиска в глубину является добавление механизма обнаружения циклов. Ни одну из вершин, уже содержащихся в пути, построенном из стартовой вершины в текущую вершину, не следует вторично рассматривать в качестве возможной альтернативы продолжения поиска. Алгоритм:

1. **Базис рекурсии**: если ***B*** – это целевая вершина, то положить ***Реш = [B]***.

1. **Шаг рекурсии**: если для исходной вершины ***B*** существует преемник ***B1***, который не принадлежит пути и из него существует ***Путь*** в целевую, то решение ***Реш = [B | Путь]***.

Это правило реализуем отношением (рис. 6.4):

**`вглубину1(Путь, Верш, Решение)`**,

где

1) ***Верш*** – исходное состояние;

2) ***Путь*** – текущий путь, строящийся из стартовой вершины;

3) ***Решение*** – ***Путь***, продолженный до целевой вершины.

Аргумент ***Путь*** позволяет не рассматривать тех преемников вершины ***Верш***, которые уже встречались раньше (обнаружение циклов).

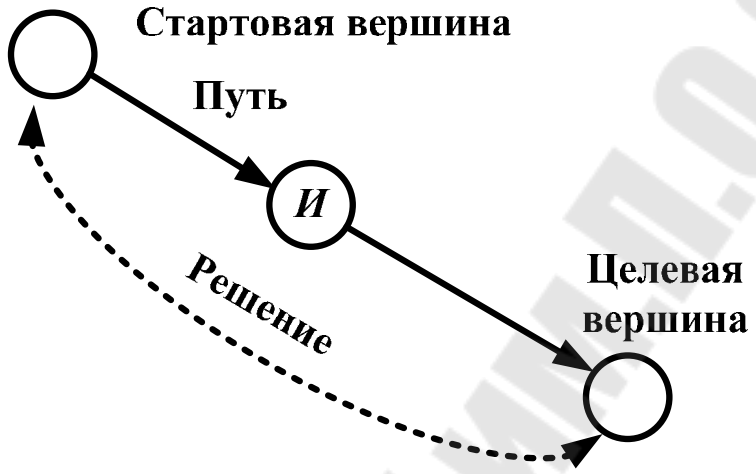



*Рис. 6.4*. Отношение в глубину1 (Путь, *В*, Решение)

**Пример 6.2.** Для пространства состояний, представленого рис. 6.1, реализовать поиск в глубину с защитой от зацикливания. Задание связей фактом *потомок – преемник*.

```
domains
list=symbol*

predicates
после(symbol, symbol)
цель(symbol)
принадлежит(symbol, list)
решить2(symbol, list)
вглубину1(list, symbol, list)

clauses
   после(a,b). после(a,c). после(b,d). после(b,e).
   после(d,h). после(h,d). после(e,i). после(e,j).
   после(e,f). после(c,f). после(c,g). после(f,k).

   решить2(Верш, Решение) :-
   вглубину1([], Верш, Решение).
```

```
   вглубину1(Путь, Верш, [Верш|Путь] ) :- цель(Верш).

   вглубину1(Путь, Верш, Реш) :-
      после(Верш, Верш1),
      not(принадлежит(Верш1, Путь)),
      вглубину1([Верш|Путь], Верш1, Реш).
   принадлежит(Name,[Name|_]).
   принадлежит(Name,[_|Tail]):-
принадлежит(Name,Tail).

   цель(h).
goal
решить2(a,T), write(T),nl, fail.
```

Вершины в списках, представляющих решение, расположены в обратном порядке: *["h","d","b","a"]*.

При поиске в бесконечном пространстве состояний существует возможность «потерять» цель, двигаясь вдоль бесконечной ветви графа так и не приблизившись к цели. Для предотвращения таких ситуаций усовершенствуем алгоритм, введя ограничение на глубину поиска. Аргумент ***МаксГлуб*** – уменьшается на единицу, при каждом рекурсивном обращений, пока не станет отрицательным.

**Пример 6.3.** Для пространства состояний, представленого рис. 6.2, реализовать поиск с ограничением глубины. Задание связей фактом *потомок – преемник*.

```
domains
list=symbol*

predicates
решить3(symbol, list, integer)
вглубину2(symbol, list, integer)

clauses
   решить3(Верш, Решение,МаксГлуб) :-
   вглубину2(Верш, Решение,МаксГлуб).

   вглубину2(Верш,[Верш], _ ) :- цель(Верш).

   вглубину2(Верш, [Верш|Реш], МаксГлуб) :-
     МаксГлуб > 0, после(Верш, Верш1),
     Макс1=МаксГлуб - 1,
     вглубину2(Верш1, Реш, Макс1).
```

```
   цель(h).

goal
решить3(a,T,3), write(T),nl, fail.
["a","c","f","k"]
решить3(a,T,4), write(T),nl, fail.
["a","b","e","f","k"]
["a","c","f","k"]
```

Для использования разработанных реализаций поиска в глубину при описании пространства состояний набором фактов вида ***связь(потомок, [приемники])***, модифицируем правило ***после***. Правило является недетерминированным (т. е. оставляет точки возврата при множестве альтернатив) и определяет всех преемников для заданной вершины (потомка). Для последовательного определения всех элементов списка воспользуемся вспомогательным предикатом:

```
элем_списка([Список]), Элемент).
```

Тогда правило **после** будет иметь следующий вид:

```
predicates
   связь(symbol, list)
   элем_списка(list, symbol)
   после(symbol, symbol)

clauses
   после(Верш,Верш1):-
   связь(Верш,Преемники),
   элем_списка(Преемники,Верш1).

   элем_списка([Преемник|_], Преемник).
   элем_списка([_|Хвост], Преемник):-
   элем_списка(Хвост, Преемник).
```

Новое описание позволяет задать пространства состояний (рис. 6.1) следующим образом:

```
связь(a, [b,c]). связь(b, [d,e]). связь(d, [h]).
связь(e, [i,j,f]). связь(c, [f,g]). связь(f, [k]).
```

## 6.3. Стратегия поиска в ширину

Стратегия поиска в ширину предусматривает переход в первую очередь к вершинам, ближайшим к стартовой. В результате процесс поиска имеет тенденцию развиваться более в ширину, чем в глубину (рис. 6.5).

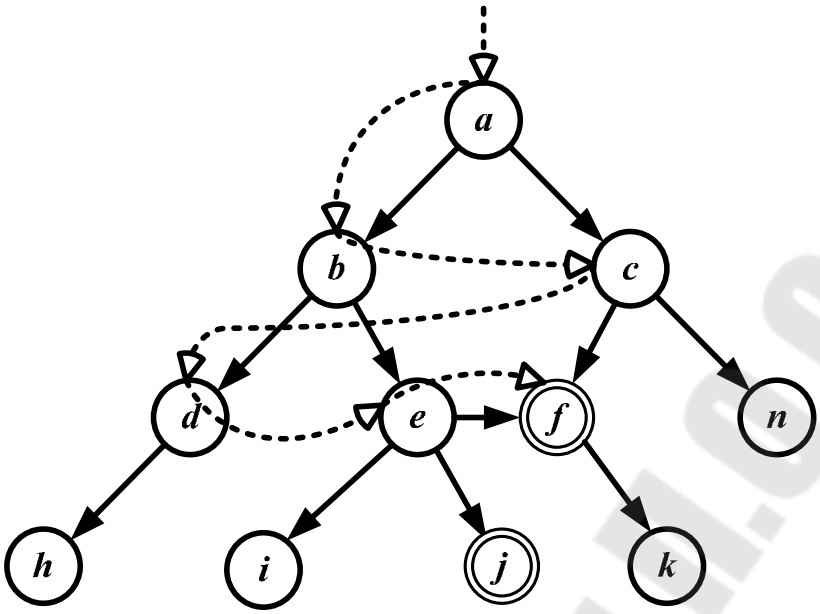



*Рис. 6.5*. Стратегия поиска в ширину для целевой вершины *f*

Программирование поиска в ширину сложнее, чем в глубину, поскольку приходится сохранять все множество альтернативных вершин – кандидатов, а не только одну вершину, как при поиске в глубину. А при определении и решающего пути требуется хранить и множество путей-кандидатов. Алгоритм поиска в ширину среди путей кандидатов:

1. **Базовое правило**: если голова первого пути – это целевая вершина, то этот путь – решение.

2. **Рекурсивное правило**: удалить первый путь из множества кандидатов и породить множество всех возможных продолжений этого пути на один шаг; множество продолжений добавить в конец множества кандидатов, а затем выполнить поиск в ширину с полученным новым множеством.

Множество путей кандидатов представляет собой список, каждый элемент которого является списком вершин. Вершины, задающие путь, располагаются в обратном порядке. Поиск начинается с одноэлементного множества кандидатов – ***[[СтартВерш]]***.

Для продолжения пути на один шаг используется встроенный предикат ***findall***, генерирующий список путей результатов ***НовПути***, которые потом добавляются в конец списка путей ***Пути***. За счет чего поиск в ширину всегда первым обнаруживает самое короткое решение, что не верно в отношении стратегии поиска в глубину.

**Пример 6.4.** Для пространства состояний, представленого рис. 6.2, реализовать поиск в ширину. Задание связей фактом *потомок – преемник.*

```
domains
   list=symbol*
   list2=list*

predicates
   решить(symbol, list)
   вширину(list2, list)
   после(symbol, symbol)
   цель(symbol)
   вспомогательный(symbol, list, list)
   принадлежит(symbol, list)
   присоединить(list2, list2, list2)

clauses
   после(a,b). после(a,c). после(b,d).
   после(b,e). после(d,h). после(e,i).
   после(e,j). после(c,f). после(c,g).
   после(f,k). после(h,d).

   решить(Старт, Решение) :-
   вширину([[Старт]], Решение).

   вширину([[Верш|Путь]|_], [Верш|Путь]) :-
       цель(Верш).
   вширину([[В|Путь]|Пути], Решение) :-
       findall(ПутьРез,вспомогательный(В,Путь,ПутьРез)
,НовПути),
       присоединить(Пути, НовПути, Пути1),
       вширину(Пути1, Решение).

   вспомогательный(В,Путь,[В1,В|Путь]):-
       после(В, В1), not(принадлежит(В1,[В|Путь])).

   принадлежит(Name,[Name|_]).
   принадлежит(Name,[_|Tail]):-
принадлежит(Name,Tail).

   присоединить([], List, List).
   присоединить([H|L1], List2, [H|L3]) :-
   присоединить(L1, List2, L3).
```

```
цель(f).

goal
решить(a,T), write(T),nl, fail.
```

Рассмотрим процесс решения для цели – ***цель(f)***.

1. Начальное множество путей кандидатов:

```
[ [a] ].
```

2. Порождаем продолжения пути [a]:

```
[ [b, a], [c, a] ].
```

3. Удаляем первый путь из множества кандидатов и порождаем его продолжения:

```
[ [d, b, a], [e, b, a] ].
```

Добавляем список продолжений в конец списка кандидатов:

```
[ [c, a], [d, b, a], [e, b, a] ].
```

4. Удаляем **[c, a]**, а затем добавляем все его продолжения в конец множества ***кандидатов***. Получаем:

```
[ [d, b, a], [e, b, a], [f, c, a], [g, c, a] ].
```

5. Далее, после того, как пути *[d, b, a]* и *[e, b, a]* будут продолжены, измененный список кандидатов примет вид:

```
[[f, c, a], [g, c, a], [h, d, b, a], [i, e, b, a],
[j, e, b, a]].
```

В данный момент обнаруживается путь *[f, c, a]*, содержащий целевую вершину *f*. Этот путь выдается в качестве решения.

# 7. МЕТОДЫ ИЗВЛЕЧЕНИЯ ЗНАНИЙ

## 7.1. Классификация методов извлечения знаний

*Извлечением знаний* называется выявление знаний из источников и преобразование их в нужную форму (например, в виде обобщенных моделей, построенных на основе методов формализации, организации и анализа знаний), а также перенос в базу знаний интеллектуальной системы. Источниками знаний могут быть книги, архивные документы, содержимое других баз знаний и т. п. Другим типом знаний являются экспертные знания, носителями которых являются специалисты (эксперты в некоторой области). Соответственно методы извлечения знаний подразделяются на текстологические и коммутативные (рис. 7.1).

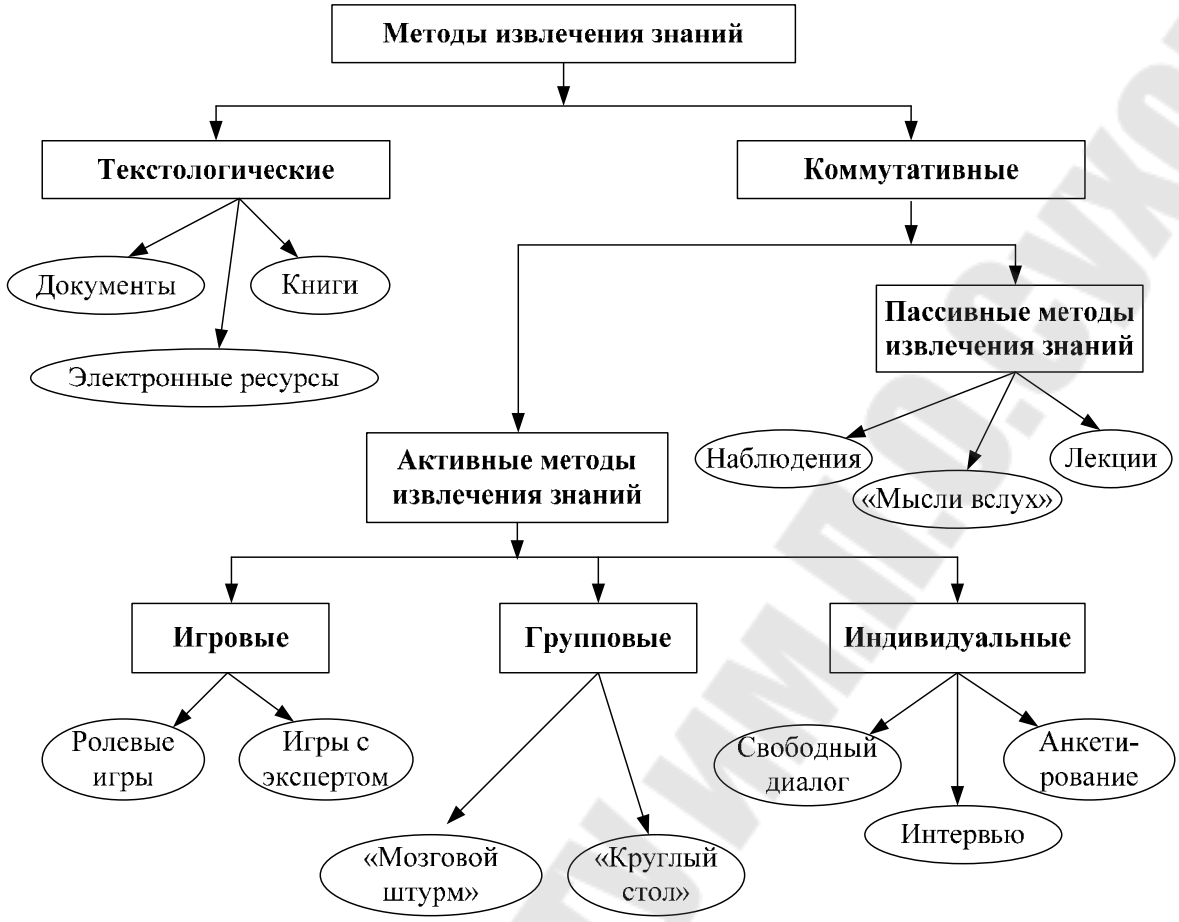



*Рис. 7.1*. Классификация методов извлечения знаний

В процессе формализации знаний выделим две основные роли. Первая роль – эксперт, т. е. человек, являющийся носителем знаний. Вторая роль – инженер по знаниям или аналитик, задачей которого является извлечение знаний из источника знаний, в качестве которого может выступать как эксперт, так и печатные или электронные материалы.

Приобретение (извлечение) и формализация знаний – это специфическая задача, направленная на выявление знаний, используемых при решении конкретной задачи в некоторой области. Поэтому основной принцип классификации методов извлечения знаний – по источнику знаний. В зависимости от источника (человек или документация) различают текстологические и коммутативные методы.

***Группа текстологических методов*** объединяет методы извлечения знаний, основанные на изучении специальных текстов из технической документации, учебников, монографий, статей и т. п. Эта группа методов является наименее изученной. В общем виде задачу извлечения знаний из текстов можно сформулировать как задачу понимания и выделения смысла текста. При этом можно выделить как

минимум две смысловые структуры: $M_1$ – смысл, который пытался заложить автор; $M_2$ – смысл, который постигает читатель (инженер по знаниям) в процессе интерпретации (рис. 7.2). Сложность процесса заключается в принципиальной невозможности совпадения знаний ($M_1 \neq M_2$). Смысл $M_1$, который закладывает автор, образуется на основе его представлений о действительности, которые невозможно изложить в одной или нескольких печатных работах. В свою очередь, $M_2$ образуется в процессе интерпретации текста на основе всей совокупности научного и личного багажа читателя. Поэтому разные читатели извлекут из одного текста различные смысловые модели.

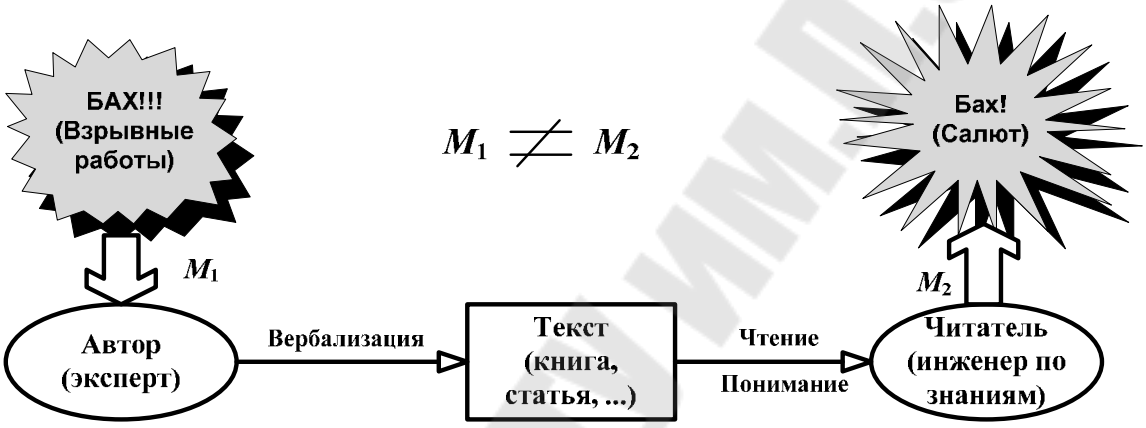



*Рис. 7.2*. Принципы текстологических методов извлечения знаний

При использовании текстологических методов извлечения знаний необходимо решить задачу декомпозиции текса на составляющие для выделения истинно значимых для реализации базы знаний фрагментов. Сложность интерпретации научных и специальных текстов заключается в том, что любой текст приобретает смысл только в контексте окружения, в которое «погружен» текст. Таким образом, для выявления смысла текста необходимо решить следующие задачи:

1) сформулировать предварительную гипотезу о смысле текста;

2) определить значения непонятных слов;

3) сформулировать общую гипотезу о содержании текста (т. е., какие знания можно извлечь из этого текста);

4) уточнить значения специфических терминов и интерпретировать фрагментов текста под влиянием общей гипотезы (от целого к частному);

5) выявить внутренние связи между отдельными (ключевыми) словами и фрагментами. Сформировать абстрактные понятия. Обобщить конкретные фрагменты знаний;

6) сформировать некоторую смысловую структуру текста;

7) скорректировать общую гипотезу относительно содержащихся в тексте фрагментов знаний (от частей к целому);

8) принять основную гипотезу, т. е. сформировать $M_2$.

На рис. 7.3 представлен упрощенный алгоритм извлечения знаний из текста, построенный на основании приведенных выше рассуждений.


*Рис. 7.3*. Алгоритм извлечения знаний из текстов

## 7.2. Коммуникативные методы извлечения знаний

***Коммуникативные методы*** охватывают способы и процедуры контактов инженера по знаниям с непосредственным источником знаний – специалистами и экспертами. Причем для эффективной работы необходимо комбинировать различные методы.

Коммуникативные методы, в свою очередь, можно разделить на две группы (рис. 7.1). *Пассивные* методы подразумевают, что ведущая роль в процедуре извлечения передается эксперту, а аналитик только наблюдает за экспертом во время его работы или записывает то, что эксперт считает нужным рассказать в форме лекции. В *активных* методах инициатива переходит к аналитику, который контактирует с экспертом в играх, диалогах, беседах «за круглым столом» и т. д. В свою очередь, *активные методы* можно разделить на две группы в зависимости от числа экспертов-специалистов – индивидуальные и групповые. *Групповые методы* обычно активизируют мышление участников дискуссий и позволяют выявлять весьма нетривиальные аспекты знаний. На сегодняшний день *индивидуальные методы* получили наибольшее распространение в силу своей простоты и минимализма. В настоящее время широкое распространение получили *игровые методы*. ***Игра*** – это особая форма деятельности (творчества), которая позволяет человеку чувствовать себя свободно (в отличие от обычной трудовой деятельности, где над человеком «висит» гнет обязательств). Рассмотрим данные методы более подробно.

В процессе *наблюдений* аналитик находится рядом с экспертом во время его профессиональной деятельности. При подготовке к сеансу извлечения знаний эксперту-специалисту необходимо объяснить цель наблюдений и попросить комментировать свои действия. Во время сеанса аналитик записывает (протоколирует) действия эксперта и его пояснения. Причем во время сеанса необходимо обеспечить невмешательство наблюдателя в работу эксперта. ***Наблюдения*** – один из наиболее распространенных методов извлечения знаний на начальных этапах разработки. Обычно он применяется в совокупности с другими методами.

Протоколирование «*мыслей вслух*» отличается от наблюдений тем, что эксперта-специалиста просят не просто прокомментировать свои действия и решения, но и объяснить, как это решение было найдено, т. е. продемонстрировать всю цепочку рассуждений. Во время рассуждений специалиста все его слова, весь «поток сознания» про-

токолируется; при этом отмечаются даже паузы и междометия. Иногда этот метод называют «*вербальные* (*словесные*) *отчеты*».

***Лекция*** – очень старый способ передачи знаний. Лекторское искусство издревле высоко ценилось во всех областях. Если эксперт имеет опыт преподавателя, то можно воспользоваться таким концентрированным фрагментом знаний, как лекция. В лекции эксперту предоставлено много степеней свободы для самовыражения; однако аналитик должен сформулировать эксперту тему и задачу лекции. Метод извлечения знаний в форме лекций, как и все пассивные методы, используют в начале разработки как эффективный способ быстрого погружения аналитика в предметную область.

*Активные индивидуальные методы* извлечения знаний на сегодняшний день наиболее распространены. В той или иной степени к ним прибегают при разработке практически любой корпоративной интеллектуальной системы. К числу основных активных методов можно отнести анкетирование, интервью, свободный диалог (специалистами компании). Во всех этих методах активную функцию выполняет аналитик, который пишет сценарий и режиссирует сеансы извлечения знаний. Первые три метода, которые можно назвать вопросными методами поиска знаний, сходны между собой и различаются лишь степенью свободы, которую может себе позволить аналитик при проведении сеансов извлечения знаний.

***Анкетирование*** – наиболее жесткий метод, поскольку он наиболее стандартизирован. В данном случае аналитик заранее составляет вопросник или анкету, размножает ее и использует для опроса нескольких работников компании.

Под ***интервью*** будем понимать специфическую форму общения аналитика и специалиста предприятия, в которой аналитик задает серию заранее подготовленных вопросов с целью извлечения знаний о предметной области. Наиболее активно интервью применяется в журналистике. Большинство специалистов в этих областях отмечают крайнюю недостаточность теоретических и методологических исследований по тематике интервьюирования.

*Свободный диалог* – это метод извлечения знаний в форме беседы аналитика и специалиста предприятия, в которой нет жесткого регламентированного плана и вопросника. Это определение не означает, что к свободному диалогу не надо готовиться. Напротив, внешне свободная и легкая форма требует серьезной профессиональной и психологической подготовки. Подготовка занимает разное время в за-

висимости от степени профессионализма аналитика, но в любом случае она необходима, т. к. несколько уменьшает вероятность самого нерационального метода – метода проб и ошибок.

К *активным групповым методам* извлечения знаний относятся дискуссии за «круглым» столом и «мозговой штурм». Основное достоинство групповых методов – это возможность одновременного поглощения знаний от нескольких экспертов, взаимодействие которых вносит элемент принципиальной новизны от наложения разных взглядов и позиций.

***Метод «круглого» стола*** предусматривает обсуждение какой-либо проблемы из выбранной предметной области, в котором принимают участие с равными правами несколько экспертов. Обычно вначале участники высказываются в определенном порядке, а затем переходят к живой свободной дискуссии. Число участников дискуссии колеблется от трех до пяти-семи. Существует специфика, связанная с поведением человека в группе.

***Мозговой штурм***, или ***мозговая атака*** – один из наиболее распространенных методов раскрепощения и активизации творческого мышления. Замечено, что боязнь критики мешает творческому мышлению, поэтому основная идея штурма – это отделение процедуры генерирования идей в замкнутой группе специалистов от процесса анализа и оценки высказанных идей. Как правило, штурм длится от 10 до 60 минут. Участникам предлагается высказывать любые идеи (научные, ненаучные, фантастические, ошибочные) на заданную тему, причем критика идей запрещена. Основной девиз штурма – «чем больше идей, тем лучше». За время штурма может быть сформировано до 100 идей. Кульминационный момент штурма – это наступление пика. В данный момент идеи начинают генерироваться с огромной скоростью, т. е. происходит непроизвольная (бессознательная) генерация гипотез участниками. Основная задача штурма – зафиксировать как можно больше идей. После штурма в спокойной обстановке проводится анализ этих идей (обычно только около 10 % идей оказываются плодотворными). Как правило, анализ идей проводят эксперты, не принимавшие участие в штурме (если это возможно).

Среди многообразия игровых методов можно выделить экспертные и ролевые игры.

***Экспертные игры*** – эксперимент, моделирующий произвольную ситуацию, где участники игры должны принимать некоторые решения. Решения анализируются и вскрываются закономерности.

Экспертные игры различаются по числу участников (индивидуальные, групповые), по виду используемого оборудования (тренажеры, компьютерная техника, наглядные материалы и т. п.). Эффективность моделирования реальных ситуаций игровыми методами сегодня подтверждена во всех областях жизнедеятельности человека. Достоинства этих методов – раскрепощение человека во время игры, выработка специфических навыков, развитие логического мышления.

**Индивидуальные игры с экспертом**. В данном случае с экспертом играет инженер по знаниям, который берет на себя какую-нибудь роль в моделируемой ситуации. Например, игра «Учитель – ученик», в которой инженер по знаниям берет на себя роль ученика и на глазах эксперта выполняет его работу, а эксперт поправляет ошибки. Игра – удобный способ раскрепостить даже застенчивого эксперта.

**Ролевые игры в группе**. Групповые игры предусматривают участие в игре нескольких экспертов. К такой игре обычно заранее составляется сценарий, распределяются роли, к каждой роли готовится портрет-описание и разрабатывается система оценивания игроков.

**Игры с тренажерами**. Игры с тренажерами в значительной степени ближе не к играм, а к имитационным упражнениям в ситуации, приближенной к действительности. Наличие тренажера позволяет воссоздать почти производственную ситуацию и понаблюдать за экспертом. Тренажеры широко применяют для обучения (например, летчиков).

**Компьютерные экспертные игры**. Компьютерные игры обычно разделяют: на позиционные игры (шахматы, шашки); динамические игры (связанные со скоростью реакции – стрельба по движущейся мишени); зрелищные или диалоговые фильмы, где пользователь может влиять на сюжет; обучающие, в которых пользователь, играя, осваивает какие-то навыки или узнает что-то новое для себя.

## 7.3. Структурирование знаний

Под *концептуальным анализом знаний* (или структурированием) понимается процесс анализа информации, полученной от источника знаний, и синтез ее (или кодирование) в некоторые структуры, не зависящие от какой-либо программной реализации. Процесс преобразования знаний от представлений эксперта к представлению в компьютере можно рассматривать как проблему преобразования информации, осуществляемой путем перехода от одного материального носителя знаний к другому.

***Поле знаний*** – это условное неформальное описание основных понятий и взаимосвязей между понятиями предметной области, выявленных из системы знаний эксперта, в виде графа, диаграммы, таблицы или текста.

В процессе извлечения знаний сначала желательно получить от эксперта поверхностные знания, постепенно переходя к глубинным структурам и более абстрактным понятиям. При формировании поля знаний необходимо учитывать особенности эмпирических знаний: модальность, противоречивость, неполнота и т. д. Аналитик должен за частным всегда видеть общее, т. е. строить цепочки «факт – обобщенный факт – эмпирический закон – теоретический закон». Центральное звено цепочки – формализация эмпирики. При этом основным на этапе формализации становится не извлечение «слепых» непонятных связей, а понимание внутренней структурной связи понятий предметной области. Искусство аналитика состоит в стремлении к созданию ясной и понятной модели проблемной области.

Следует также учитывать, что эксперты в проблемной области не всегда опираются на логические рассуждения. В их представлениях о проблемной области и методах решения задач, характерных для нее, широкое применение находят ассоциативные рассуждения и рассуждения правдоподобия. Рассмотрим алгоритм формирования поля знаний, предложенный Т. А. Гавриловой [7]:

1. *Определение входных и выходных данных*. Этот шаг определяет направление движения в поле знаний – от входных данных к выходным. Кроме того, структура входных и выходных данных существенно влияет на форму и содержание поля знаний. На этом шаге определение может быть достаточно размытым, в дальнейшем оно будет уточняться.

2. *Составление словаря терминов и наборов ключевых слов*. На этом шаге проводится текстуальный анализ всех протоколов сеансов извлечения знаний. Выписываются все значимые слова, обозначающие понятия, явления, процессы, предметы, действия, признаки и т. д. При этом следует попытаться разобраться в значении терминов. Важен осмысленный словарь.

3. *Выявление объектов и понятий*. Производится «просеивание» словаря и выбор значимых понятий и их признаков. В идеале на этом шаге образуется полный систематический набор терминов из какой-либо области знаний.

4. *Выявление связей между понятиями* (или *построение сети ассоциаций*). Все в мире взаимосвязано. На данном этапе необходимо определить, как направлены связи, что ближе, а что дальше, что первично, а что вторично. Таким образом, строится сеть ассоциаций, где связи только намечены, но пока не поименованы.

5. *Выявление метапонятий и детализации поня*тий. Связи, полученные на предыдущем шаге, позволяют инженеру по знаниям структурировать понятия и как выявлять понятия более высокого уровня обобщения (метапонятия), так и детализировать на более низком уровне.

6. *Построение пирамиды знаний, или иерархической лестницы понятий*. Под *пирамидой знаний* будем понимать иерархическую лестницу понятий, подъем по которой означает углубление понимания и повышения уровня абстракции (обобщенности) понятий. Количество уровней в пирамиде зависит от особенностей предметной области, профессионализма экспертов и инженеров по знаниям.

7. *Определение отношений между понятиями*. Эти отношения выявляются как внутри каждого из уровней пирамиды, так и между уровнями. Фактически на этом шаге даются имена тем связям, которые обнаруживаются на шагах 4 и 5, а также обозначаются причинно-следственные, лингвистические, временные и другие виды отношений.

8. *Определение стратегий принятия решений*. Определение стратегий принятия решения, т. е. выявление цепочек рассуждений, связывает все сформированные ранее понятия и отношения в динамическую систему поля знаний. Именно стратегии придают активность знаниям, именно они «перетряхивают» модель в поисках пути от входных данных к результату.

## Литература

1. Адаменко, А. Н. Логическое программирование и Visual Prolog / А. Н. Адаменко, А. М. Кучков. – СПб. : БХВ-Петербург, 2003. – 992 с.

2. Барендрегт, Х. Лямбда-исчисление: его синтаксис и семантика / Х. Барендрегт. – М. : Мир, 1985.

3. Барсегян, А. А. Технологии анализа данных / А. А. Барсегян [и др.]. – СПб. : БХВ-Петербург, 2007. – 384 с.

4. Барский, А. Б. Нейронные сети: распознавание, управление, принятие решений / А. Б. Барский. – М. : Финансы и статистика, 2004. – 176 с.

5. Братко, И. Алгоритмы искусственного интеллекта на языке PROLOG / И. Братко. – М. : Вильямс, 2004. – 640 с.

6. Братко, И. Программирование на языке Пролог для искусственного интеллекта / И. Братко. – М. : Мир, 1990.

7. Гаврилова, Т. А. Базы знаний интеллектуальных систем / Т. А. Гаврилова, В. Ф. Хорошевский. – СПб. : Питер, 2000. – 384 с.

8. Герман, О. В. Введение в теорию экспертных систем и обработку знаний / О. В. Герсан. – Минск : ДизайнПРО, 1995.

9. Городняя, Л. В. Основы функционального программирования : курс лекций : учеб. пособие. – М. : ИНТУИТ.РУ «Интернет-университет информационных технологий», 2004.

10. Девятков, В. В. Системы искусственного интеллекта : учеб. пособие для вузов. – М. : МГТУ им. Н. Э. Баумана, 2001. – 352 с.

11. Джексон, П. Введение в экспертные системы / П. Джексон. – Вильямс, 2001.

12. Душкин, Р. В. Функциональное программирование на языке Haskell / Р. В. Душкин. – М. : ДМК Пресс, 2007. – 608 с.

13. Искусственный интеллект. В 3 кн. Кн. 2. Модели и методы : справочник / под ред. Д. А. Поспелова. – М. : Радио и связь, 1990.

14. Полещук, Н. Н. AutoLISP и Visual LISP в среде AutoCAD / Н. Н. Полещук, П. В. Лоскутов. – СПб. : БХВ-Петербург, 2006. – 960 с.

15. Рыжиков, Ю. И. Имитационное моделирование. Теория и технологии / Ю. И. Рыжиков. – СПб. : КОРОНАпринт. – М. : Альтекс-А, 2004. – 384 с.

16. Смолин, Д. В. Введение в искусственный интеллект : конспект лекций / Д. В. Смолин. – М. : ФИЗМАТЛИТ, 2004. – 208 с.

17. Стерлинг, Л. Искусство программирования на языке Пролог / Л. Стерлинг, Э. Шапиро. – М. : Мир, 1990. – 235 с.

18. Хювенен, Э. Мир Лиспа. В 2 т. / Э. Хювенен, И. Сеппянен. – М. : Мир, 1990.

19. Seibel, P. Practical Common Lisp / P. Seibel. – APress, 2005. – 528 p.

Учебное электронное издание комбинированного распространения

Учебное издание

**Мурашко** Игорь Александрович
**Литвинов** Дмитрий Александрович

# БАЗЫ ЗНАНИЙ И ПОДДЕРЖКА ПРИНЯТИЯ РЕШЕНИЙ В СИСТЕМАХ АВТОМАТИЗИРОВАННОГО ПРОЕКТИРОВАНИЯ

## Курс лекций
## по одноименной дисциплине для студентов специальности 1-40 01 02 «Информационные системы и технологии (по направлениям)» дневной формы обучения

**Электронный аналог печатного издания**

Редактор *Н. Г. Мансурова*
Компьютерная верстка *Н. Б. Козловская*

Подписано в печать 30.09.11.
Формат 60x84/16. Бумага офсетная. Гарнитура «Таймс».
Ризография. Усл. печ. л. 4,88. Уч.-изд. л. 5,1.
Изд. № 16.
E-mail: ic@gstu.by
http://www.gstu.by

Издатель и полиграфическое исполнение:
Издательский центр учреждения образования
«Гомельский государственный технический университет
имени П. О. Сухого».
ЛИ № 02330/0549424 от 08.04.2009 г.
246746, г. Гомель, пр. Октября, 48